VISIT…

LANZAROTE
Caliente.COM

Contents

Contents viii

Illustrations xix

Preface xxxviii

1 Introduction to Computers, the Internet, the Web and C# 1

1.1 Introduction 2
1.2 What Is a Computer? 3
1.3 Computer Organization 3
1.4 Evolution of Operating Systems 4
1.5 Personal Computing, Distributed Computing and Client/Server Computing 5
1.6 Machine Languages, Assembly Languages and High-level Languages 6
1.7 C, C++, Visual Basic .NET and Java™ 7
1.8 C# 9
1.9 Other High-level Languages 10
1.10 Structured Programming 10
1.11 Key Software Trend: Object Technology 11
1.12 Hardware Trends 13
1.13 History of the Internet and World Wide Web 13
1.14 World Wide Web Consortium (W3C) 15
1.15 Extensible Markup Language (XML) 15
1.16 Introduction to Microsoft .NET 16
1.17 .NET Framework and the Common Language Runtime 18
1.18 Tour of the Book 20
1.19 Internet and World Wide Web Resources 29

2 Introduction to the Visual Studio .NET IDE 33
2.1 Introduction 34
2.2 Visual Studio .NET Integrated Development Environment (IDE) Overview 34
2.3 Menu Bar and Toolbar 37
2.4 Visual Studio .NET Windows 39
    2.4.1 **Solution Explorer** 39
    2.4.2 **Toolbox** 40
    2.4.3 **Properties** Window 42
2.5 Using Help 42
2.6 Simple Program: Displaying Text and an Image 44

3 Introduction to C# Programming 59
3.1 Introduction 60
3.2 Simple Program: Printing a Line of Text 60
3.3 Another Simple Program: Adding Integers 71
3.4 Memory Concepts 75
3.5 Arithmetic 76
3.6 Decision Making: Equality and Relational Operators 80

4 Control Structures: Part 1 94
4.1 Introduction 95
4.2 Algorithms 95
4.3 Pseudocode 96
4.4 Control Structures 96
4.5 **`if`** Selection Structure 99
4.6 **`if`**/**`else`** Selection Structure 100
4.7 **`while`** Repetition Structure 105
4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition) 106
4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition) 109
4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures) 116
4.11 Assignment Operators 120
4.12 Increment and Decrement Operators 121
4.13 Introduction to Windows Application Programming 124

5 Control Structures: Part 2 139
5.1 Introduction 140
5.2 Essentials of Counter-Controlled Repetition 140
5.3 **`for`** Repetition Structure 142
5.4 Examples Using the **`for`** Structure 146
5.5 **`switch`** Multiple-Selection Structure 152
5.6 **`do`**/**`while`** Repetition Structure 156
5.7 Statements **`break`** and **`continue`** 158
5.8 Logical and Conditional Operators 160
5.9 Structured-Programming Summary 166

6 Methods 178
6.1 Introduction 179
6.2 Program Modules in C# 179
6.3 `Math` Class Methods 181
6.4 Methods 181
6.5 Method Definitions 183
6.6 Argument Promotion 193
6.7 C# Namespaces 195
6.8 Value Types and Reference Types 196
6.9 Passing Arguments: Pass-by-Value vs. Pass-by-Reference 197
6.10 Random-Number Generation 200
6.11 Example: Game of Chance 207
6.12 Duration of Variables 212
6.13 Scope Rules 212
6.14 Recursion 215
6.15 Example Using Recursion: The Fibonacci Series 219
6.16 Recursion vs. Iteration 222
6.17 Method Overloading 223

7 Arrays 236
7.1 Introduction 237
7.2 Arrays 237
7.3 Declaring and Allocating Arrays 239
7.4 Examples Using Arrays 240
  7.4.1 Allocating an Array and Initializing Its Elements 240
  7.4.2 Totaling the Elements of an Array 242
  7.4.3 Using Histograms to Display Array Data Graphically 243
  7.4.4 Using the Elements of an Array as Counters 244
  7.4.5 Using Arrays to Analyze Survey Results 247
7.5 Passing Arrays to Methods 250
7.6 Passing Arrays by Value and by Reference 253
7.7 Sorting Arrays 257
7.8 Searching Arrays: Linear Search and Binary Search 260
  7.8.1 Searching an Array with Linear Search 260
  7.8.2 Searching a Sorted Array with Binary Search 261
7.9 Multiple-Subscripted Arrays 265
7.10 `foreach` Repetition Structure 272

8 Object-Based Programming 280
8.1 Introduction 281
8.2 Implementing a Time Abstract Data Type with a Class 282
8.3 Class Scope 290
8.4 Controlling Access to Members 290
8.5 Initializing Class Objects: Constructors 292
8.6 Using Overloaded Constructors 293
8.7 Properties 297

8.8 Composition: Objects References as Instance Variables of Other Classes 306
8.9 Using the **`this`** Reference 309
8.10 Garbage Collection 311
8.11 **`static`** Class Members 312
8.12 **`const`** and **`readonly`** Members 317
8.13 Indexers 319
8.14 Data Abstraction and Information Hiding 326
8.15 Software Reusability 327
8.16 Namespaces and Assemblies 328
8.17 **Class View** and **Object Browser** 333

9 Object-Oriented Programming: Inheritance 342
9.1 Introduction 343
9.2 Base Classes and Derived Classes 344
9.3 **`protected`** and **`internal`** Members 347
9.4 Relationship between Base Classes and Derived Classes 347
9.5 Case Study: Three-Level Inheritance Hierarchy 368
9.6 Constructors and Destructors in Derived Classes 371
9.7 Software Engineering with Inheritance 377

10 Object-Oriented Programming: Polymorphism 382
10.1 Introduction 383
10.2 Derived-Class-Object to Base-Class-Object Conversion 383
10.3 Type Fields and **`switch`** Statements 390
10.4 Polymorphism Examples 391
10.5 Abstract Classes and Methods 392
10.6 Case Study: Inheriting Interface and Implementation 394
10.7 **`sealed`** Classes and Methods 402
10.8 Case Study: Payroll System Using Polymorphism 403
10.9 Case Study: Creating and Using Interfaces 413
10.10 Delegates 425
10.11 Operator Overloading 430

11 Exception Handling 438
11.1 Introduction 439
11.2 Exception Handling Overview 440
11.3 Example: **`DivideByZeroException`** 443
11.4 .NET **`Exception`** Hierarchy 448
11.5 **`finally`** Block 449
11.6 **`Exception`** Properties 457
11.7 Programmer-Defined Exception Classes 462
11.8 Handling Overflows with Operators **`checked`** and **`unchecked`** 466

12 Graphical User Interface Concepts: Part 1 474
12.1 Introduction 475
12.2 Windows Forms 476

12.3 Event-Handling Model 479
    12.3.1 Basic Event Handling 480
12.4 Control Properties and Layout 484
12.5 `Label`s, `TextBox`es and `Button`s 488
12.6 `GroupBox`es and `Panel`s 495
12.7 `CheckBox`es and `RadioButton`s 498
12.8 `PictureBox`es 507
12.9 Mouse Event Handling 509
12.10 Keyboard Event Handling 511

13 Graphical User Interfaces Concepts: Part 2 520

13.1 Introduction 521
13.2 Menus 521
13.3 `LinkLabel`s 530
13.4 `ListBox`es and `CheckedListBox`es 534
    13.4.1 `ListBox`es 537
    13.4.2 `CheckedListBox`es 539
13.5 `ComboBox`es 542
13.6 `TreeView`s 547
13.7 `ListView`s 553
13.8 Tab Control 560
13.9 Multiple-Document-Interface (MDI) Windows 565
13.10 Visual Inheritance 574
13.11 User-Defined Controls 578

14 Multithreading 590

14.1 Introduction 591
14.2 Thread States: Life Cycle of a Thread 592
14.3 Thread Priorities and Thread Scheduling 594
14.4 Thread Synchronization and Class `Monitor` 599
14.5 Producer/Consumer Relationship without Thread Synchronization 601
14.6 Producer/Consumer Relationship with Thread Synchronization 607
14.7 Producer/Consumer Relationship: Circular Buffer 616

15 Strings, Characters and Regular Expressions 632

15.1 Introduction 633
15.2 Fundamentals of Characters and Strings 633
15.3 `String` Constructors 635
15.4 `String` `Indexer`, `Length` Property and `CopyTo` Method 636
15.5 Comparing `String`s 638
15.6 `String` Method `GetHashCode` 642
15.7 Locating Characters and Substrings in `String`s 643
15.8 Extracting Substrings from `String`s 646
15.9 Concatenating `String`s 647
15.10 Miscellaneous `String` Methods 648

15.11 Class **`StringBuilder`** 650
15.12 **`StringBuilder`** Indexer, **`Length`** and **`Capacity`** Properties, and **`EnsureCapacity`** Method 652
15.13 **`StringBuilder`** **`Append`** and **`AppendFormat`** Methods 654
15.14 **`StringBuilder`** **`Insert`**, **`Remove`** and **`Replace`** Methods 658
15.15 **`Char`** Methods 661
15.16 Card Shuffling and Dealing Simulation 664
15.17 Regular Expressions and Class **`Regex`** 668

16 Graphics and Multimedia 684
16.1 Introduction 685
16.2 Graphics Contexts and Graphics Objects 687
16.3 Color Control 688
16.4 Font Control 696
16.5 Drawing Lines, Rectangles and Ovals 701
16.6 Drawing Arcs 704
16.7 Drawing Polygons and Polyli]nes 707
16.8 Advanced Graphics Capabilities 711
16.9 Introduction to Multimedia 717
16.10 Loading, Displaying and Scaling Images 718
16.11 Animating a Series of Images 720
16.12 Windows Media Player 733
16.13 Microsoft Agent 736

17 Files and Streams 757 6
17.1 Introduction 757
17.2 Data Hierarchy 757
17.3 Files and Streams 759
17.4 Classes **`File`** and **`Directory`** 761
17.5 Creating a Sequential-Access File 771
17.6 Reading Data from a Sequential-Access File 783
17.7 Random-Access Files 794
17.8 Creating a Random-Access File 798
17.9 Writing Data Randomly to a Random-Access File 802
17.10 Reading Data Sequentially from a Random-Access File 807
17.11 Case Study: A Transaction-Processing Program 812

18 Extensible Markup Language (XML) 838
18.1 Introduction 839
18.2 XML Documents 839
18.3 XML Namespaces 844
18.4 Document Object Model (DOM) 847
18.5 Document Type Definitions (DTDs), Schemas and Validation 865
  18.5.1 Document Type Definitions 866
  18.5.2 Microsoft XML Schemas 869
  18.5.3 W3C XML Schema 872

18.5.4 Schema Validation in C# 873
18.6 Extensible Stylesheet Language and **XslTransform** 877
18.7 Microsoft BizTalk™ 884
18.8 Internet and World Wide Web Resources 887

19 Database, SQL and ADO .NET 895

19.1 Introduction 896
19.2 Relational Database Model 897
19.3 Relational Database Overview: **Books** Database 898
19.4 Structured Query Language (SQL) 905
19.4.1 Basic **SELECT** Query 905
19.4.2 **WHERE** Clause 906
19.4.3 **ORDER BY** Clause 909
19.4.4 Merging Data from Multiple Tables: **INNER JOIN** 912
19.4.5 Joining Data from Tables **Authors**, **AuthorISBN**, **Titles** and **Publishers** 914
19.4.6 **INSERT** Statement 917
19.4.7 **UPDATE** Statement 918
19.4.8 **DELETE** Statement 919
19.5 ADO .NET Object Model 920
19.6 Programming with ADO .NET: Extracting Information from a DBMS 921
19.6.1 Connecting to and Querying an Access Data Source 921
19.6.2 Querying the **Books** Database 928
19.7 Programming with ADO.NET: Modifying a DBMS 930
19.8 Reading and Writing XML Files 938

20 ASP .NET, Web Forms and Web Controls 948

20.1 Introduction 949
20.2 Simple HTTP Transaction 950
20.3 System Architecture 952
20.4 Creating and Running a Simple Web Form Example 953
20.5 Web Controls 966
20.5.1 Text and Graphics Controls 967
20.5.2 **AdRotator** Control 971
20.5.3 Validation Controls 976
20.6 Session Tracking 987
20.6.1 Cookies 988
20.6.2 Session Tracking with **HttpSessionState** 997
20.7 Case Study: Online Guest Book 1006
20.8 Case Study: Connecting to a Database in ASP .NET 1013
20.9 Tracing 1027
20.10 Internet and World Wide Web Resources 1030

21 ASP .NET and Web Services 1039

21.1 Introduction 1040

21.2 Web Services 1041
21.3 Simple Object Access Protocol (SOAP) and Web Services 1044
21.4 Publishing and Consuming Web Services 1046
21.5 Session Tracking in Web Services 1062
21.6 Using Web Forms and Web Services 1075
21.7 Case Study: Temperature Information Application 1081
21.8 User-Defined Types in Web Services 1091

22 Networking: Streams-Based Sockets and Datagrams 1106
22.1 Introduction 1107
22.2 Establishing a Simple Server (Using Stream Sockets) 1108
22.3 Establishing a Simple Client (Using Stream Sockets) 1110
22.4 Client/Server Interaction with Stream-Socket Connections 1111
22.5 Connectionless Client/Server Interaction with Datagrams 1120
22.6 Client/Server Tic-Tac-Toe Using a Multithreaded Server 1125

23 Data Structures and Collections 1145
23.1 Introduction 1146
23.2 Self-Referential Classes 1146
23.3 Linked Lists 1148
23.4 Stacks 1160
23.5 Queues 1165
23.6 Trees 1168
    23.6.1 Binary Search Tree of Integer Values 1170
    23.6.2 Binary Search Tree of `IComparable` Objects 1177
23.7 Collection Classes 1185
    23.7.1 Class `Array` 1185
    23.7.2 Class `ArrayList` 1188
    23.7.3 Class `Stack` 1194
    23.7.4 Class `Hashtable` 1198

24 Accessibility 1212
24.1 Introduction 1213
24.2 Regulations and Resources 1214
24.3 Web Accessibility Initiative 1216
24.4 Providing Alternatives for Images 1216
24.5 Maximizing Readability by Focusing on Structure 1218
24.6 Accessibility in Visual Studio .NET 1218
    24.6.1 Enlarging Toolbar Icons 1219
    24.6.2 Enlarging the Text 1220
    24.6.3 Modifying the Toolbox 1221
    24.6.4 Modifying the Keyboard 1221
    24.6.5 Rearranging Windows 1222
24.7 Accessibility in C# 1224
24.8 Accessibility in XHTML Tables 1230
24.9 Accessibility in XHTML Frames 1234

24.10 Accessibility in XML 1235
24.11 Using Voice Synthesis and Recognition with VoiceXML™ 1235
24.12 CallXML™ 1243
24.13 JAWS® for Windows 1248
24.14 Other Accessibility Tools 1249
24.15 Accessibility in Microsoft® Windows® 2000 1251
  24.15.1 Tools for People with Visual Impairments 1252
  24.15.2 Tools for People with Hearing Impairments 1254
  24.15.3 Tools for Users Who Have Difficulty Using the Keyboard 1255
  24.15.4 Microsoft Narrator 1258
  24.15.5 Microsoft On-Screen Keyboard 1261
  24.15.6 Accessibility Features in Microsoft Internet Explorer 5.5 1262
24.16 Internet and World Wide Web Resources 1264

A Operator Precedence Chart 1273

B Number Systems (on CD) 1275
B.1 Introduction 1276
B.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers 1279
B.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers 1281
B.4 Converting from Binary, Octal or Hexadecimal to Decimal 1281
B.5 Converting from Decimal to Binary, Octal, or Hexadecimal 1282
B.6 Negative Binary Numbers: Two’s Complement Notation 1283

C Career Opportunities (on CD) 1289
C.1 Introduction 1290
C.2 Resources for the Job Seeker 1291
C.3 Online Opportunities for Employers 1292
C.4 Recruiting Services 1297
C.5 Career Sites 1298
C.6 Internet and World Wide Web Resources 1303

D Visual Studio .NET Debugger 1311
D.1 Introduction 1312
D.2 Breakpoints 1313
D.3 Examining Data 1315
D.4 Program Control 1318
D.5 Additional Method Debugging Capabilities 1322
D.6 Additional Class Debugging Capabilities 1324

E Generating Documentation in Visual Studio (on CD) 1329
E.1 Introduction 1330
E.2 Documentation Comments 1330
E.3 Documenting C# Source Code 1331

E.4 Creating Comment Web Pages 1339
E.5 Creating XML Documentation Files 1341

F ASCII Character Set 1348

G Unicode® (on CD) 1349
G.1 Introduction 1350
G.2 Unicode Transformation Formats 1351
G.3 Characters and Glyphs 1352
G.4 Advantages and Disadvantages of Unicode 1353
G.5 Unicode Consortium’s Web Site 1353
G.6 Using Unicode 1355
G.7 Character Ranges 1357

H COM Integration (on CD) 1362
H.1 Introduction 1362
H.2 ActiveX Integration 1364
H.3 DLL Integration 1367
H.4 Internet and World Wide Web Resources 1371

I Introduction to HyperText Markup Language 4: Part 1 (on CD) 1374
I.1 Introduction 1375
I.2 Markup Languages 1375
I.3 Editing HTML 1376
I.4 Common Elements 1376
I.5 Headers 1379
I.6 Linking 1380
I.7 Images 1382
I.8 Special Characters and More Line Breaks 1386
I.9 Unordered Lists 1388
I.10 Nested and Ordered Lists 1389
I.11 Internet and World Wide Web Resources 1392

J Introduction to HyperText Markup Language 4: Part 2 (on CD) 1397
J.1 Introduction 1398
J.2 Basic HTML Tables 1398
J.3 Intermediate HTML Tables and Formatting 1400
J.4 Basic HTML Forms 1403
J.5 More Complex HTML Forms 1406
J.6 Internal Linking 1413
J.7 Creating and Using Image Maps 1416
J.8 `<meta>` Tags 1418
J.9 `frameset` Element 1420

J.10 Nested **`framesets`** 1422
J.11 Internet and World Wide Web Resources 1424

K Introduction to XHTML: Part 1 (on CD) 1430
K.1 Introduction 1431
K.2 Editing XHTML 1431
K.3 First XHTML Example 1432
K.4 W3C XHTML Validation Service 1435
K.5 Headers 1436
K.6 Linking 1438
K.7 Images 1441
K.8 Special Characters and More Line Breaks 1445
K.9 Unordered Lists 1447
K.10 Nested and Ordered Lists 1448
K.11 Internet and World Wide Web Resources 1451

L Introduction to XHTML: Part 2 (on CD) 1456
L.1 Introduction 1457
L.2 Basic XHTML Tables 1457
L.3 Intermediate XHTML Tables and Formatting 1460
L.4 Basic XHTML Forms 1462
L.5 More Complex XHTML Forms 1465
L.6 Internal Linking 1473
L.7 Creating and Using Image Maps 1476
L.8 **`meta`** Elements 1478
L.9 **`frameset`** Element 1479
L.10 Nested **`framesets`** 1483
L.11 Internet and World Wide Web Resources 1485

M HTML/XHTML Special Characters 1491

N HTML/XHTML Colors 1492

O Bit Manipulation (on CD) 1495
O.1 Introduction 1496
O.2 Bit Manipulation and the Bitwise Operators 1496
O.3 Class **`BitArray`** 1508

P Crystal Reports® for Visual Studio .NET 1513
P.1 Introduction 1513
P.2 Crystal Reports Web Site Resources 1513
P.3 Crystal Reports and Visual Studio .NET 1514

Bibliography 1518

Index 1522

Illustrations

1 Introduction to Computers, the Internet, the Web and C#

1.1 .NET Languages . 19

2 Introduction to the Visual Studio .NET IDE

2.1 **Start Page** in Visual Studio .NET. 35
2.2 **New Project** dialog. 36
2.3 Visual Studio .NET environment after a new project has been created. 37
2.4 Visual Studio .NET menu bar. 38
2.5 Visual Studio .NET menu summary. 38
2.6 Visual Studio .NET toolbar. 38
2.7 Tool tip demonstration. 39
2.8 Toolbar icons for various Visual Studio .NET windows. 39
2.9 **Solution Explorer** window. 40
2.10 **Toolbox** window. 41
2.11 Demonstrating window auto-hide. 41
2.12 **Properties** window. 43
2.13 **Dynamic Help** window. 44
2.14 Simple program as it executes. 45
2.15 Creating a new Windows application. 45
2.16 Setting the project location. 46
2.17 Setting the form's **`Text`** property. 46
2.18 Form with sizing handles. *47*
2.19 Changing property **`BackColor`**. 47
2.20 Adding a new label to the form. 48
2.21 Label in position with its **`Text`** property set. 48
2.22 **Properties** window displaying the label's properties. 49
2.23 **Font** window for selecting fonts, styles and sizes. 49
2.24 Centering the text in the label. 50

2.25 Inserting and aligning the picture box. 50
2.26 **`Image`** property of the picture box. 51
2.27 Selecting an image for the picture box. 51
2.28 Picture box after the image has been inserted. 51
2.29 IDE in run mode, with the running application in the foreground. 52

3 Introduction to C# Programming

3.1 Our first program in C#. 60
3.2 Visual Studio .NET-generated console application. 65
3.3 Execution of the **`Welcome1`** program. 66
3.4 Printing on one line with separate statements. 67
3.5 Printing on multiple lines with a single statement. 67
3.6 Some common escape sequences. 68
3.7 Displaying multiple lines in a dialog. 68
3.8 Adding a reference to an assembly in Visual Studio .NET. 70
3.9 Internet Explorer's GUI. 71
3.10 Dialog displayed by calling **`MessageBox.Show`**. 71
3.11 Addition program that adds two values entered by the user. 72
3.12 Memory location showing name and value of variable **`number1`**. 75
3.13 Memory locations after values for variables **`number1`** and **`number2`** have been input. 76
3.14 Memory locations after a calculation. 76
3.15 Arithmetic operators. 77
3.16 Precedence of arithmetic operators. 78
3.17 Order in which a second-degree polynomial is evaluated. 80
3.18 Equality and relational operators. 81
3.19 Using equality and relational operators. 81
3.20 Precedence and associativity of operators discussed in this chapter. 85

4 Control Structures: Part 1

4.1 Flowcharting C#'s sequence structure. 97
4.2 C# keywords. 98
4.3 Flowcharting a single-selection **`if`** structure. 100
4.4 Flowcharting a double-selection **`if`**/**`else`** structure. 101
4.5 Flowcharting the **`while`** repetition structure. 106
4.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem. 107
4.7 Class average program with counter-controlled repetition. 107
4.8 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem. 111
4.9 Class-average program with sentinel-controlled repetition. 112
4.10 Pseudocode for examination-results problem. 118
4.11 C# program for examination-results problem. 118
4.12 Arithmetic assignment operators. 120
4.13 The increment and decrement operators. 121
4.14 The difference between preincrementing and postincrementing. 122
4.15 Precedence and associativity of the operators discussed so far in this book. 123

4.16 IDE showing program code for Fig. 2.15. 125
4.17 Windows Form Designer generated code when expanded. 126
4.18 Code generated by the IDE for `welcomeLabel`. 127
4.19 Using the **Properties** window to set a property value. 127
4.20 Windows Form Designer generated code reflecting new property values. 128
4.21 Changing a property in the code view editor. 128
4.22 New `Text` property value reflected in design mode. 129
4.23 Method `FrmASimpleProgram_Load`. 129
4.24 Changing a property value at runtime. 130

5 Control Structures: Part 2

5.1 Counter-controlled repetition with `while` structure. 141
5.2 Counter-controlled repetition with the `for` structure. 142
5.3 Components of a typical `for` header. 143
5.4 Flowcharting a typical `for` repetition structure. 146
5.5 Summation using `for`. 147
5.6 Icons for message dialogs. 148
5.7 Buttons for message dialogs. 148
5.8 Calculating compound interest with `for`. 149
5.9 `string` formatting codes. 151
5.10 `switch` multiple-selection structure. 152
5.11 Flowcharting the `switch` multiple-selection structure. 155
5.12 `do`/`while` repetition structure. 156
5.13 Flowcharting the `do`/`while` repetition structure. 157
5.14 `break` statement in a `for` structure. 158
5.15 `continue` statement in a `for` structure. 159
5.16 Truth table for the && (conditional AND) operator. 161
5.17 Truth table for the || (conditional OR) operator. 162
5.18 Truth table for the logical exclusive OR (^) operator. 163
5.19 Truth table for operator `!` (logical NOT). 163
5.20 Conditional and logical operators. 164
5.21 Precedence and associativity of the operators discussed so far. 166
5.22 C#'s single-entry/single-exit sequence, selection and repetition structures. 167
5.23 Rules for forming structured programs. 168
5.24 Simplest flowchart. 168
5.25 Repeatedly applying rule 2 of Fig. 5.23 to the simplest flowchart. 169
5.26 Applying rule 3 of Fig. 5.23 to the simplest flowchart. 169
5.27 Stacked, nested and overlapped building blocks. 170
5.28 Unstructured flowchart. 170

6 Methods

6.1 Hierarchical boss method/worker method relationship. 180
6.2 Commonly used `Math` class methods. 182
6.3 Using programmer-defined method `Square`. 183
6.4 Programmer-defined `Maximum` method. 188
6.5 Allowed implicit conversions. 194
6.6 Namespaces in the Framework Class Library. 195

6.7 C# built-in data types. 196
6.8 Demonstrating **`ref`** and **`out`** parameters. 198
6.9 Random integers in the range 1–6. 201
6.10 Rolling dice in a Windows application . 203
6.11 Simulating rolling 12 six-sided dice. 205
6.12 Program to simulate the game of craps. 208
6.13 Scoping. 213
6.14 Recursive evaluation of 5!. 217
6.15 Calculating factorials with a recursive method. 217
6.16 Recursively generating Fibonacci numbers. 219
6.17 Set of recursive calls to method **`Fibonacci`**. 221
6.18 Using overloaded methods. 223
6.19 Syntax error generated from overloaded methods with identical parameter lists and different return types. 225
6.20 The Towers of Hanoi for the case with four disks. 234

7 Arrays

7.1 A 12-element array. 238
7.2 Precedence and associativity of the operators discussed so far. 239
7.3 Initializing element arrays in three different ways. 240
7.4 Computing the sum of the elements of an array. 242
7.5 Program that prints histograms. 243
7.6 Using arrays to eliminate a **`switch`** structure. 245
7.7 Simple student-poll analysis program. 248
7.8 Passing arrays and individual array elements to methods. 251
7.9 Passing an array reference by value and by reference . 254
7.10 Sorting an array with bubble sort. 257
7.11 Linear search of an array. 260
7.12 Binary search of a sorted array. 262
7.13 Double-subscripted array with three rows and four columns. 266
7.14 Initializing multidimensional arrays. 267
7.15 Example using double-subscripted arrays. 270
7.16 Using **`For Each`**/**`Next`** with an array. 272

8 Object-Based Programming

8.1 **`Time1`** abstract data type represents the time in 24-hour format. 283
8.2 Using an abstract data type. 287
8.3 Accessing **`private`** class members from client code generates syntax errors. 291
8.4 Overloaded constructors provide flexible object-initialization options. 293
8.5 Overloaded constructor demonstration. 295
8.6 Properties provide controlled access to an object’s data. 298
8.7 Properties demonstration for class **`Time3`**. 301
8.8 **`Date`** class encapsulates day, month and year information. 306
8.9 **`Employee`** class encapsulates employee name, birthday and hire date. 308
8.10 Composition demonstration. 309

8.11 **`this`** reference used implicitly and explicitly to enable an object to manipulate its own data and invoke its own methods. (Part 1 of 2) 310
8.12 **`this`** reference demonstration. 311
8.13 **`static`** members are accessible to all objects of a class. 314
8.14 **`static`** member demonstration. 315
8.15 **`const`** and **`readonly`** class member demonstration. 318
8.16 Indexers provide subscripted access to an object's members. 320
8.17 Assembly **`TimeLibrary`** contains class **`Time3`**. 329
8.18 Simple Class Library. 332
8.19 Assembly **`TimeLibrary`** used from class **`AssemblyTest`**. 333
8.20 **Class View** of class **`Time1`** (Fig. 8.1) and class **`TimeTest`** (Fig. 8.2). 334
8.21 **Object Browser** when user selects **`Object`** from **`Time1.cs`**. 335

9 Object-Oriented Programming: Inheritance

9.1 Inheritance examples. 345
9.2 Inheritance hierarchy for university **`CommunityMember`**s. 346
9.3 Portion of a **`Shape`** class hierarchy. 347
9.4 **`Point`** class represents an *x*-*y* coordinate pair. 348
9.5 **`PointTest`** class demonstrates class **`Point`** functionality. 350
9.6 **`Circle`** class contains an *x*-*y* coordinate and a radius. 351
9.7 **`CircleTest`** demonstrates class **`Circle`** functionality. 354
9.8 **`Circle2`** class that inherits from class **`Point`**. 355
9.9 **`Point2`** class represents an *x*-*y* coordinate pair as **`protected`** data. 357
9.10 **`Circle3`** class that inherits from class **`Point2`**. 359
9.11 **`CircleTest3`** demonstrates class **`Circle3`** functionality. 360
9.12 **`Point3`** class uses properties to manipulate its **`private`** data. 362
9.13 **`Circle4`** class that inherits from class **`Point3`**, which does not provide **`protected`** data. 364
9.14 **`CircleTest4`** demonstrates class **`Circle4`** functionality. 366
9.15 **`Cylinder`** class inherits from class **`Circle4`** and overrides method **`Area`**. 368
9.16 Testing class **`Cylinder`**. 370
9.17 **`Point4`** base class contains constructors and finalizer. 372
9.18 **`Circle5`** class inherits from class **`Point3`** and overrides a finalizer method. 374
9.19 Order in which constructors and destructors are called. 376

10 Object-Oriented Programming: Polymorphism

10.1 **`Point`** class represents an *x*-*y* coordinate pair. 384
10.2 **`Circle`** class that inherits from class **`Point`**. 385
10.3 Assigning derived-class references to base-class references. 387
10.4 Abstract **`Shape`** base class. 394
10.5 **`Point2`** class inherits from **`abstract`** class **`Shape`**. 395
10.6 **`Circle2`** class that inherits from class **`Point2`**. 397
10.7 **`Cylinder2`** class inherits from class **`Circle2`**. 398
10.8 **`AbstractShapesTest`** demonstrates polymorphism in Point-Circle-Cylinder hierarchy. 400

10.9 **abstract** class **Employee** definition. 404
10.10 **Boss** class inherits from class **Employee**. 405
10.11 **CommissionWorker** class inherits from class **Employee**. 406
10.12 **PieceWorker** class inherits from class **Employee**. 408
10.13 **HourlyWorker** class inherits from class **Employee** . 410
10.14 **EmployeesTest** class tests the **Employee** class hierarchy. 412
10.15 Interface for returning age of objects of disparate classes. 415
10.16 **Person** class implements **IAge** interface. 415
10.17 **Tree** class implements **IAge** interface. 416
10.18 Demonstrate polymorphism on objects of disparate classes. 417
10.19 **IShape** interface provides methods **Area** and **Volume** and property **Name**. 419
10.20 **Point3** class implements interface **IShape**. 420
10.21 **Circle3** class inherits from class **Point3**. 421
10.22 **Cylinder3** class inherits from class **Circle3**. 423
10.23 **Interfaces2Test** uses interfaces to demonstrate polymorphism in Point-Circle-Cylinder hierarchy. 424
10.24 Bubble sort using delegates. 426
10.25 Bubble-sort **Form** application. 427
10.26 Overloading operators for complex numbers. 430
10.27 Using operator overloading. 433

11 Exception Handling

11.1 Exception handlers for **FormatException** and **DivideByZeroException**. 443
11.2 Demonstrating that **finally** blocks always execute regardless of whether or not an exception occurs. 452
11.3 **Exception** properties and stack unwinding. 458
11.4 **ApplicationException** subclass thrown when a program performs illegal operations on negative numbers. 463
11.5 **SquareRootTest** class thrown an exception if error occurs when calculating the square root. 464
11.6 Operators **checked** and **unchecked** and the handling of arithmetic overflow. 467

12 Graphical User Interface Concepts: Part 1

12.1 Sample Internet Explorer window with GUI components. 476
12.2 Some basic GUI components. 476
12.3 Components and controls for Windows Forms. 477
12.4 Common **Form** properties and events. 478
12.5 Event-handling model using delegates. 479
12.6 Events section of the **Properties** window. 480
12.7 Simple event-handling example using visual programming. 481
12.8 List of **Form** events. 482
12.9 Details of **Click** event. 483
12.10 Class **Control** properties and methods. 485
12.11 Anchoring demonstration. 486

12.12 Manipulating the **Anchor** property of a control. 487
12.13 Docking demonstration. 487
12.14 Class **Control** layout properties. 487
12.15 **Label** properties. 489
12.16 **TextBox** properties and events. (Part 1 of 2) 489
12.17 **Button** properties and events. 490
12.18 Program to display hidden text in a password box. 490
12.19 **GroupBox** properties. 495
12.20 **Panel** properties. 495
12.21 Creating a **Panel** with scrollbars. 496
12.22 Using **GroupBox**es and **Panel**s to arrange **Button**s. 496
12.23 **CheckBox** properties and events. 498
12.24 Using **CheckBox**es to change font styles. 499
12.25 **RadioButton** properties and events. 502
12.26 Using **RadioButton**s to set message-window options. 502
12.27 **PictureBox** properties and events. 507
12.28 Using a **PictureBox** to display images. 507
12.29 Mouse events, delegates and event arguments. 509
12.30 Using the mouse to draw on a form. 510
12.31 Keyboard events, delegates and event arguments. 512
12.32 Demonstrating keyboard events . 513
12.33 GUI for Exercise 12.4. 519
12.34 GUI for Exercise 12.5. 519
12.35 GUI for Exercise 12.6. 519

13 Graphical User Interfaces Concepts: Part 2

13.1 Expanded and checked menus. 522
13.2 Visual Studio .NET **Menu Designer**. 523
13.3 **MainMenu** and **MenuItem** properties and events. 524
13.4 Menus for changing text font and color. 525
13.5 **LinkLabel** control in the design phase and in running program. 531
13.6 **LinkLabel** properties and events. 531
13.7 **LinkLabel**s used to link to a folder, a Web page and an application. 532
13.8 **ListBox** and **CheckedListBox** on a form. 535
13.9 **ListBox** properties, methods and events. 535
13.10 **String Collection Editor**. 537
13.11 **ListBox** used in a program to add, remove and clear items. 537
13.12 **CheckedListBox** properties, methods and events. 540
13.13 **CheckedListBox** and **ListBox** used in a program to display a user selection. 541
13.14 **ComboBox** demonstration. 543
13.15 **ComboBox** properties and events. 543
13.16 **ComboBox** used to draw a selected shape. 544
13.17 **TreeView** displaying a sample tree. 548
13.18 **TreeView** properties and events. 548
13.19 **TreeNode** properties and methods. 549

13.20 **TreeNode Editor**. 550
13.21 **`TreeView`** used to display directories. 550
13.22 **`ListView`** properties and events. 554
13.23 **Image Collection Editor** window for an **`ImageList`** component. 555
13.24 **`ListView`** displaying files and folders. 555
13.25 Tabbed pages in Visual Studio .NET. 560
13.26 **`TabControl`** with **`TabPage`**s example. 561
13.27 **`TabPage`**s added to a **`TabControl`**. 561
13.28 **`TabControl`** properties and events. 561
13.29 **`TabControl`** used to display various font settings. 562
13.30 MDI parent window and MDI child windows. 566
13.31 SDI and MDI forms. 566
13.32 MDI parent and MDI child events and properties. 567
13.33 Minimized and maximized child windows. 568
13.34 **`MenuItem`** property **`MdiList`** example. 569
13.35 **`LayoutMdi`** enumeration values. 570
13.36 MDI parent-window class. 571
13.37 **`Child`** class for MDI demonstration. 574
13.38 Class **`FrmInheritance`**, which inherits from class **`Form`**, contains a button (**Learn More**). 575
13.39 Visual Inheritance through the Form Designer. 577
13.40 Class **`FrmVisualTest`**, which inherits from class **`VisualForm.FrmInheritance`**, contains an additional button. 577
13.41 Custom control creation. 579
13.42 Programmer-defined control that displays the current time. 580
13.43 Custom-control creation. 582
13.44 Project properties dialog. 582
13.45 Custom control added to the **ToolBox**. 582
13.46 Custom control added to a **`Form`**. 583

14 Multithreading

14.1 Thread life cycle. 593
14.2 Thread-priority scheduling. 596
14.3 Threads sleeping and printing. 596
14.4 Producer and consumer threads accessing a shared object without synchronization. 602
14.5 Producer and consumer threads accessing a shared object with synchronization. 607
14.6 Producer and consumer threads accessing a circular buffer. 617

15 Strings, Characters and Regular Expressions

15.1 **`String`** constructors. 635
15.2 **`String`** indexer, **`Length`** properties and **`CopyTo`** method. 636
15.3 **`String`** test to determine equality. 639
15.4 **`StartsWith`** and **`EndsWith`** methods. 641
15.5 **`GetHashCode`** method demonstration. 642
15.6 Searching for characters and substrings in **`string`**s. 643

15.7 Substrings generated from **strings**. 646
15.8 **Concat static** method. 648
15.9 **String** methods **Replace**, **ToLower**, **ToUpper**, **Trim** and **ToString**. 649
15.10 **StringBuilder** class constructors. 651
15.11 **StringBuilder** size manipulation. 653
15.12 Append methods of **StringBuilder**. 655
15.13 **StringBuilder**’s **AppendFormat** method. 656
15.14 **StringBuilder** text insertion and removal. 658
15.15 **StringBuilder** text replacement. 659
15.16 **Char**’s **static** character-testing methods and case-conversion methods. 661
15.17 **Card** class. 664
15.18 Card dealing and shuffling simulation. 665
15.19 Character classes. 669
15.20 Regular expressions checking birthdays. 669
15.21 Quantifiers used regular expressions. 672
15.22 Validating user information using regular expressions. 672
15.23 **Regex** methods **Replace** and **Split**. 677

16 Graphics and Multimedia

16.1 **System.Drawing** namespace’s classes and structures. 686
16.2 GDI+ coordinate system. Units are measured in pixels. 687
16.3 **Color** structure **static** constants and their RGB values. 689
16.4 **Color** structure members . 689
16.5 Classes that derive from class **Brush**. 690
16.6 Color value and alpha demonstration. 690
16.7 **ColorDialog** used to change background and text color. 693
16.8 **Font** class read-only properties. 696
16.9 **Font**s and **FontStyle**s. 697
16.10 An illustration of font metrics. 699
16.11 **FontFamily** methods that return font-metric information. 699
16.12 **FontFamily** class used to obtain font-metric information. 699
16.13 **Graphics** methods that draw lines, rectangles and ovals. 702
16.14 Demonstration of methods that draw lines, rectangles and ellipses. 702
16.15 Ellipse bounded by a rectangle. 704
16.16 Positive and negative arc angles. 705
16.17 **Graphics** methods for drawing arcs. 705
16.18 Arc-method demonstration. 705
16.19 **Graphics** methods for drawing polygons. 708
16.20 Polygon-drawing demonstration. 708
16.21 Shapes drawn on a form. 712
16.22 Paths used to draw stars on a form. 715
16.23 Image resizing. 718
16.24 Animation of a series of images. 720
16.25 Container class for chess pieces. 723
16.26 Chess-game code. 725
16.27 Windows Media Player demonstration. 733

16.28 Peedy introducing himself when the window opens. 737
16.29 Peedy's *Pleased* animation. 738
16.30 Peedy's reaction when he is clicked. 738
16.31 Peedy flying animation. 739
16.32 Peedy waiting for speech input. 740
16.33 Peedy repeating the user's request for Seattle-style pizza. 740
16.34 Peedy repeating the user's request for anchovies as an additional topping. 741
16.35 Peedy recounting the order. 741
16.36 Peedy calculating the total. 742
16.37 Microsoft Agent demonstration. 742
16.38 GUI for Eight Queens exercise. 755

17 Files and Streams

17.1 Data hierarchy. 759
17.2 C#'s view of an *n-byte* file. 760
17.3 **`File`** class methods (partial list). 761
17.4 **`Directory`** class methods (partial list). 761
17.5 Testing classes **`File`** and **`Directory`**. 762
17.6 Regular expression used to determine file types. 766
17.7 Base class for GUIs in our file-processing applications. 772
17.8 Record for sequential-access file-processing applications. 774
17.9 Create and write to a sequential-access file. 777
17.10 Sample data for the program of Fig. 17.9. 783
17.11 Reading sequential-access files. 784
17.12 Credit-inquiry program. 788
17.13 Random-access file with fixed-length records. 795
17.14 Record for random-access file-processing applications. 795
17.15 Creating files for random-access file-processing applications. 799
17.16 Writing records to random-access files. 802
17.17 Reading records from random-access files sequentially. 807
17.18 Record-transaction class for the transaction-processor case study. 812
17.19 **`TransactionProcessorForm`** class runs the transaction-processor application. 816
17.20 **`StartDialogForm`** class enables users to access dialog boxes associated with various transactions. 817
17.21 **`UpdateDialogForm`** class enables users to update records in transaction-processor case study. 822
17.22 **`NewDialogForm`** class enables users to create records in transaction-processor case study. 827
17.23 **`DeleteDialogForm`** class enables users to remove records from files in transaction-processor case study. 830
17.24 Inventory of a hardware store. 837

18 Extensible Markup Language (XML)

18.1 XML used to mark up an article. 839
18.2 **`article.xml`** displayed by Internet Explorer. 842
18.3 XML to mark up a business letter. 842

18.4 XML namespaces demonstration. 844
18.5 Default namespaces demonstration. 846
18.6 Tree structure for Fig. 18.1. 847
18.7 **`XmlNodeReader`** used to iterate through an XML document. 848
18.8 DOM structure of an XML document illustrated by a class . 851
18.9 **`XPathNavigator`** class used to navigate selected nodes. 858
18.10 XML document that describes various sports . 864
18.11 XPath expressions and descriptions. 865
18.12 Document Type Definition (DTD) for a business letter. 866
18.13 XML document referencing its associated DTD. 867
18.14 XML Validator validates an XML document against a DTD. 869
18.15 XML Validator displaying an error message. 870
18.16 XML document that conforms to a Microsoft Schema document. 870
18.17 Microsoft Schema file that contains structure to which **`bookxdr.xml`** conforms. 871
18.18 XML document that conforms to W3C XML Schema. 872
18.19 XSD Schema document to which **`bookxsd.xml`** conforms. 872
18.20 Schema-validation example. 874
18.21 XML document that does not conform to the XSD schema of Fig. 18.19. 876
18.22 XML file that does not conform to the Schema in Fig. 18.17. 876
18.23 XML document containing book information. 878
18.24 XSL document that transforms **`sorting.xml`** (Fig. 18.23) into XHTML. 879
18.25 XSL style sheet applied to an XML document. 882
18.26 BizTalk terminology. 885
18.27 BizTalk markup using an offer Schema. 885

19 Database, SQL and ADO .NET

19.1 Relational-database structure of an **`Employee`** table. 897
19.2 Result set formed by selecting **`Department`** and **`Location`** data from the **`Employee`** table. 898
19.3 **`Authors`** table from **`Books`**. 898
19.4 Data from the **`Authors`** table of **`Books`** . 899
19.5 **`Publishers`** table from **`Books`**. 899
19.6 Data from the **`Publishers`** table of **`Books`**. 899
19.7 **`AuthorISBN`** table from **`Books`**. 900
19.8 Data from **`AuthorISBN`** table in **`Books`**. 900
19.9 **`Titles`** table from **`Books`**. 901
19.10 Data from the **`Titles`** table of **`Books`**. 901
19.11 Table relationships in **`Books`**. 904
19.12 SQL query keywords. 905
19.13 **`authorID`** and **`lastName`** from the **`Authors`** table. 906
19.14 Titles with copyrights after 1999 from table **`Titles`**. 907
19.15 Authors from the **`Authors`** table whose last names start with **`D`**. 908
19.16 Authors from table **`Authors`** whose last names contain **`i`** as the second letter. 909
19.17 Authors from table **`Authors`** in ascending order by **`lastName`**. 909

19.18 Authors from table **`Authors`** in descending order by **`lastName`**. 910
19.19 Authors from table **`Authors`** in ascending order by **`lastName`** and by **`firstName`**. 911
19.20 Books from table **`Titles`** whose titles end with **`How to Program`** in ascending order by **`title`**. 912
19.21 Authors from table **`Authors`** and ISBN numbers of the authors' books, sorted in ascending order by **`lastName`** and **`firstName`**. 913
19.22 **`TitleAuthor`** query of **`Books`** database. 914
19.23 Portion of the result set produced by the query in Fig. 19.22. 915
19.24 **`Authors`** after an **`INSERT`** operation to add a record. 917
19.25 Table **`Authors`** after an **`UPDATE`** operation to change a record. 919
19.26 Table **`Authors`** after a **`DELETE`** operation to remove a record. 920
19.27 Accessing and displaying a database's data. 921
19.28 Execute SQL statements on a database. 928
19.29 Modifying a database. 930
19.30 Application that writes an XML representation of a **`DataSet`** to a file. 939
19.31 XML document generated from **`DataSet`** in **`DatabaseXMLWriter`**. 941

20 ASP .NET, Web Forms and Web Controls

20.1 Web server/client interaction. Step 1: The **`GET`** request, **`GET /books/downloads.htm HTTP/1.1`**. 951
20.2 Client interacting with Web server. Step 2: The HTTP response, **`HTTP/1.1 200 OK`**. 951
20.3 Three-tier architecture. 952
20.4 ASPX page that displays the Web server's time. 953
20.5 Code-behind file for a page that displays the Web server's time. 955
20.6 HTML response when the browser requests **`WebTime.aspx`**. 958
20.7 Creating an **ASP.NET Web Application** in Visual Studio. 960
20.8 Visual Studio creating and linking a virtual directory for the **`WebTime`** project folder. 961
20.9 **Solution Explorer** window for project **`WebTime`**. 961
20.10 **Web Forms** menu in the **Toolbox**. 962
20.11 **Design** mode of Web Form designer. 962
20.12 **HTML** mode of Web Form designer. 963
20.13 Code-behind file for **`WebForm1.aspx`** generated by Visual Studio .NET. 964
20.14 **`GridLayout`** and **`FlowLayout`** illustration. 965
20.15 **`WebForm.aspx`** after adding two **`Label`**s and setting their properties. 966
20.16 Web controls commonly used in ASP.NET applications. 967
20.17 Web controls demonstration. 967
20.18 **`AdRotator`** class demonstrated on a Web form . 972
20.19 Code-behind file for page demonstrating the **`AdRotator`** class. 973
20.20 **`AdvertisementFile`** used in **`AdRotator`** example. 974
20.21 Validators used in a Web Form that generates possible letter combinations from a phone number. 977
20.22 Code-behind file for the word-generator page. 979
20.23 HTML and ECMAScript sent to the client browser. 984

20.24 ASPX file that presents a list of programming languages. 989
20.25 Code-behind file that writes cookies to the client. 991
20.26 ASPX page that displays book information. 994
20.27 Cookies being read from a client in an ASP .NET application. 995
20.28 **`HttpCookie`** properties. 997
20.29 Options supplied on an ASPX page. 997
20.30 Sessions are created for each user in an ASP .NET Web application. 999
20.31 **`HttpSessionState`** properties. 1003
20.32 Session information displayed in a **`ListBox`**. 1003
20.33 Session data read by an ASP .NET Web application to provide recommendations for the user. 1004
20.34 Guest-book application GUI. 1007
20.35 ASPX file for the guest book application. 1007
20.36 Code-behind file for the guest book application. 1009
20.37 Log in Web Form. 1013
20.38 ASCX code for the header. 1015
20.39 Code-behind file for the log-in page of authors application. 1016
20.40 ASPX file that allows a user to select an author from a drop-down list. 1021
20.41 Database information input into a **`DataGrid`** . 1023
20.42 ASPX page with tracing turned off. 1028
20.43 Tracing enabled on a page. 1029
20.44 Tracing information for a project. 1029

21 ASP .NET and Web Services

21.1 ASMX file rendered in Internet Explorer. 1042
21.2 Service description for a Web service. 1043
21.3 Invoking a method of a Web service from a Web browser. 1044
21.4 Results of invoking a Web-service method from a Web browser. 1044
21.5 SOAP request for the **`HugeInteger`** Web service. 1045
21.6 **`HugeInteger`** Web service. 1046
21.7 Design view of a Web service. 1054
21.8 Adding a Web service reference to a project. 1055
21.9 **Add Web Reference** dialog. 1055
21.10 Web services located on **`localhost`**. 1056
21.11 Web reference selection and description. 1056
21.12 **Solution Explorer** after adding a Web reference to a project. 1057
21.13 Using the **`HugeInteger`** Web service. 1057
21.14 **`Blackjack`** Web service. 1063
21.15 Blackjack game that uses **`Blackjack`** Web service. 1067
21.16 Airline reservation Web service. 1076
21.17 Airline Web Service in design view. 1078
21.18 ASPX file that takes reservation information. 1078
21.19 Code-behind file for the reservation page. 1079
21.20 **`TemperatureServer`** Web service. 1082
21.21 Class that stores weather information about a city. 1085
21.22 Receiving temperature and weather data from a Web service. 1087

21.23 Class that stores equation information. 1093
21.24 Web service that generates random equations . 1095
21.25 Returning an object from a Web-service method. 1097
21.26 Math tutor application. 1098

22 Networking: Streams-Based Sockets and Datagrams

22.1 Server portion of a client/server stream-socket connection. 1111
22.2 Client portion of a client/server stream-socket connection. 1114
22.3 Server-side portion of connectionless client/server computing. 1120
22.4 Client portion of connectionless client/server computing. 1122
22.5 Server side of client/server Tic-Tac-Toe program. 1126
22.6 Client side of client/server Tic-Tac-Toe program. 1132
22.7 Class **Square**. 1139
22.8 English letters of the alphabet and decimal digits as expressed in international Morse code. 1144

23 Data Structures and Collections

23.1 Sample self-referential **Node** class definition. 1147
23.2 Two self-referential class objects linked together. 1148
23.3 A graphical representation of a linked list. 1149
23.4 Definitions of classes **ListNode**, **List** and **EmptyListException**. 1151
23.5 Demonstrating the linked list. 1155
23.6 A graphical representation of the **InsertAtFront** operation. 1158
23.7 A graphical representation of the **InsertAtBack** operation. 1158
23.8 A graphical representation of the **RemoveFromFront** operation. 1159
23.9 A graphical representation of the **RemoveFromBack** operation. 1160
23.10 **StackInheritance** extends class **List**. 1161
23.11 Using class **StackInheritance**. 1162
23.12 **StackComposition** class encapsulates functionality of class **List**. 1164
23.13 **QueueInheritance** extends class **List** . 1166
23.14 Using inheritance to create a queue. 1167
23.15 A graphical representation of a binary tree. 1169
23.16 A binary search tree containing 12 values. 1169
23.17 Definitions of **TreeNode** and **Tree** for a binary search tree. 1170
23.18 Creating and traversing a binary tree. 1174
23.19 A binary search tree. 1176
23.20 Definitions of class **TreeNode** and **Tree** for manipulating **IComparable** objects. 1178
23.21 Demonstrating class Tree with **IComparable** objects. 1181
23.22 Program that demonstrates class **Array** . 1186
23.23 Some methods of class **ArrayList**. 1189
23.24 Demomstrating the **ArrayList** class. 1189
23.25 Using the **Stack** class . 1195
23.26 Using the **Hashtable** class. 1200

24 Accessibility

24.1 Acts designed to improve Internet and computer accessibility for people with disabilities. 1214
24.2 We Media’s home page. (Courtesy of WeMedia, Inc.) 1215
24.3 Enlarging icons using the **Customize** feature. 1219
24.4 Enlarged icons in the development window. 1219
24.5 Text Editor before modifying the font size. 1220
24.6 Enlarging text in the **Options** window. 1220
24.7 Text Editor after the font size is modified. 1221
24.8 Adding tabs to the **Toolbox**. 1222
24.9 Shortcut key creation. 1223
24.10 Removing tabs from Visual Studio environment. 1223
24.11 Console windows with tabs and without tabs. 1224
24.12 Properties of class `Control` related to accessibility. 1225
24.13 Application with accessibility features. 1226
24.14 XHTML table without accessibility modifications. 1231
24.15 Table optimized for screen reading, using attribute `headers`. 1232
24.16 Home page written in VoiceXML. 1236
24.17 Publication page of Deitel and Associates’ VoiceXML page. 1238
24.18 VoiceXML tags. 1242
24.19 **Hello World** CallXML example. (Courtesy of Voxeo, © Voxeo Corporation 2000–2001.) 1243
24.20 CallXML example that reads three ISBN values. (Courtesy of Voxeo, © Voxeo Corporation 2000–2001.) 1245
24.21 CallXML elements. 1248
24.22 **Display Settings** dialog. 1251
24.23 **Accessibility Wizard** initialization options. 1252
24.24 Scroll Bar and Window Border Size dialog. 1252
24.25 Adjusting window-element sizes. 1253
24.26 **Display Color Settings** options. 1253
24.27 **Accessibility Wizard** mouse cursor adjustment tool. 1254
24.28 **SoundSentry** dialog. 1254
24.29 **ShowSounds** dialog. 1255
24.30 **StickyKeys** window. 1255
24.31 **BounceKeys** dialog. 1256
24.32 **ToggleKeys** window. 1256
24.33 **Extra Keyboard Help** dialog. 1257
24.34 **MouseKeys** window. 1257
24.35 **Mouse Button Settings** window. 1258
24.36 **Mouse Speed** dialog. 1259
24.37 **Set Automatic Timeouts** dialog. 1259
24.38 Saving new accessibility settings. 1260
24.39 **Narrator** window. 1260
24.40 Voice-settings window. 1261
24.41 **Narrator** reading **Notepad** text. 1261
24.42 Microsoft **On-Screen Keyboard**. 1262

24.43 Microsoft Internet Explorer 5.5's accessibility options. 1262
24.44 Advanced accessibility settings in Microsoft Internet Explorer 5.5. 1263

A Operator Precedence Chart

A.1 Operator precedence chart. 1273

B Number Systems (on CD)

B.1 Digits of the binary, octal, decimal and hexadecimal number systems. 1277
B.2 Comparison of the binary, octal, decimal and hexadecimal number systems. 1278
B.3 Positional values in the decimal number system. 1278
B.4 Positional values in the binary number system. 1278
B.5 Positional values in the octal number system. 1279
B.6 Positional values in the hexadecimal number system. 1279
B.7 Decimal, binary, octal, and hexadecimal equivalents . 1279
B.8 Converting a binary number to decimal. 1281
B.9 Converting an octal number to decimal. 1281
B.10 Converting a hexadecimal number to decimal. 1282

C Career Opportunities (on CD)

C.1 `Monster.com` home page. (Courtesy of `Monster.com`.] 1292
C.2 `FlipDog.com` job search. (Courtesy of `Flipdog.com`.) 1293
C.3 List of a job seeker's criteria. 1295
C.4 Advantage Hiring, Inc.'s Net-Interview™ service. (Courtesy of Advantage Hiring, Inc.) 1298
C.5 `eLance.com` request for proposal (RFP) example. (Courtesy of eLance, Inc.] 1301

D Visual Studio .NET Debugger

D.1 Syntax error. 1312
D.2 Debug sample program. 1313
D.3 Setting a breakpoint. 1314
D.4 Debug configuration setting. 1315
D.5 Console application suspended for debugging. 1315
D.6 Execution suspended at a breakpoint. 1316
D.7 **Watch** window. 1316
D.8 **Autos**, **Locals** and **This** windows. 1317
D.9 **Immediate** window. 1319
D.10 **Debug** toolbar icons. 1319
D.11 **Breakpoints** window. 1319
D.12 Disabled breakpoint. 1320
D.13 **New Breakpoint** dialog. 1321
D.14 **Breakpoint Hit Count** dialog. 1321
D.15 **Breakpoint Condition** dialog. 1321
D.16 Debugging methods. 1322
D.17 **Call Stack** window. 1323

D.18 IDE displaying a method’s calling point. 1323
D.19 Debug program control features. 1324
D.20 Using the **Immediate** window to debug methods. 1324
D.21 Object debugging example. 1325
D.22 Breakpoint location for class debugging. 1326
D.23 Expanded class in **Watch** window. 1327
D.24 Expanded array in **Watch** window. 1327

E Generating Documentation in Visual Studio (on CD)

E.1 `Point` marked up with XML comments. 1331
E.2 `Circle` class marked up with XML comments. 1333
E.3 `CircleTest` class marked up with XML comments. 1336
E.4 Selecting the **Build Comment Web Pages** from **Tools** menu. 1339
E.5 Saving a document to a file. 1340
E.6 XHTML documentation of class `Circle`. 1340
E.7 XHTML documentation of method `Area` method of class `Circle`. 1341
E.8 XML documentation generated by Visual Studio .NET. 1341

F ASCII Character Set

F.1 ASCII character set. 1348

G Unicode® (on CD) 1349

G.1 Correlation between the three encoding forms. 1352
G.2 Various glyphs of the character A. 1352
G.3 Unicode values for multiple languages. 1354
G.4 Some character ranges. 1357

H COM Integration (on CD)

H.1 ActiveX control registration. 1363
H.2 **Customize Toolbox** dialog with an ActiveX control selected. 1364
H.3 IDE’s toolbox and `LabelScrollbar` properties. 1365
H.4 ActiveX COM control integration in C#. 1365
H.5 **Add Reference** dialog DLL Selection. 1368
H.6 COM DLL component in C#. 1369

I Introduction to HyperText Markup Language 4: Part 1 (on CD)

I.1 Basic HTML file. 1377
I.2 Header elements `h1` through `h6`. 1379
I.3 Linking to other Web pages. 1380
I.4 Linking to an email address. 1381
I.5 Placing images in HTML files. 1382
I.6 Using images as link anchors. 1384
I.7 Inserting special characters into HTML. 1386
I.8 Unordered lists in HTML. 1388

I.9 Nested and ordered lists in HTML. 1389

J Introduction to HyperText Markup Language 4: Part 2 (on CD)

J.1 HTML table. 1398
J.2 Complex HTML table. 1401
J.3 Simple form with hidden fields and a text box. 1403
J.4 Form including textareas, password boxes and checkboxes. 1406
J.5 Form including radio buttons and pulldown lists. 1409
J.6 Using internal hyperlinks to make your pages more navigable. 1413
J.7 Picture with links anchored to an image map. 1416
J.8 Using `meta` to provide keywords and a description. 1418
J.9 Web site using two frames—navigation and content. 1420
J.10 Framed Web site with a nested frameset. 1423

K Introduction to XHTML: Part 1 (on CD)

K.1 First XHTML example. 1432
K.2 Validating an XHTML document. (Courtesy of World Wide Web Consortium (W3C).) 1434
K.3 XHTML validation results. (Courtesy of World Wide Web Consortium (W3C).) 1436
K.4 Header elements `h1` through `h6`. 1437
K.5 Linking to other Web pages. 1438
K.6 Linking to an e-mail address. 1440
K.7 Placing images in XHTML files. 1441
K.8 Using images as link anchors. 1443
K.9 Inserting special characters into XHTML. 1445
K.10 Nested and ordered lists in XHTML. 1448

L Introduction to XHTML: Part 2 (on CD)

L.1 XHTML table. 1457
L.2 Complex XHTML table. 1460
L.3 Simple form with hidden fields and a textbox. 1463
L.4 Form with textareas, password boxes and checkboxes. 1466
L.5 Form including radio buttons and drop-down lists. 1469
L.6 Using internal hyperlinks to make pages more easily navigable. 1473
L.7 Image with links anchored to an image map. 1476
L.8 Using `meta` to provide keywords and a description. 1478
L.9 Web document containing two frames—navigation and content. 1480
L.10 XHTML document displayed in the left frame of Fig. L.5. 1482
L.11 Framed Web site with a nested frameset. 1484
L.12 XHTML table for Exercise L.7. 1489
L.13 XHTML table for Exercise L.8. 1490

M HTML/XHTML Special Characters

M.1 XHTML special characters. 1491

N HTML/XHTML Colors

N.1 HTML/XHTML standard colors and hexadecimal RGB values. 1492
N.2 XHTML extended colors and hexadecimal RGB values . 1493

O Bit Manipulation (on CD)

O.1 Bitwise operators. 1497
O.2 Results of combining two bits with the bitwise AND operator (&). 1497
O.3 Results of combining two bits with the bitwise inclusive OR operator (|). 1497
O.4 Results of combining two bits with the bitwise exclusive OR operator (^). 1498
O.5 Displaying the bit representation of an integer. 1498
O.6 Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators. 1501
O.7 Using the bitshift operators. 1505
O.8 Bitwise assignment operators. 1507
O.9 Sieve of Eratosthenes. 1508

P Crystal Reports® for Visual Studio .NET

P.1 Report expert choices. (Courtesy Crystal Decisions) 1514
P.2 Expert formatting menu choices. (Courtesy of Crystal Decisions) 1515
P.3 Crystal Reports designer interface. (Courtesy of Crystal Decisions) 1516

Preface

Live in fragments no longer. Only connect.
Edward Morgan Forster

We wove a web in childhood,
A web of sunny air.
Charlotte Brontë

Welcome to C# and the world of Windows, Internet and World-Wide-Web programming with Visual Studio and the .NET platform! This book is the second in our new *.NET How to Program* series, which presents various leading-edge computing technologies in the context of the .NET platform.

C# is the next phase in the evolution of C and C++ and was developed expressly for Microsoft's .NET platform. C# provides the features that are most important to programmers, such as object-oriented programming, strings, graphics, graphical-user-interface (GUI) components, exception handling, multithreading, multimedia (audio, images, animation and video), file processing, prepackaged data structures, database processing, Internet and World-Wide-Web-based client/server networking and distributed computing. The language is appropriate for implementing Internet- and World-Wide-Web-based applications that seamlessly integrate with PC-based applications.

The .NET platform offers powerful capabilities for software development and deployment, including independence from a specific language or platform. Rather than requiring developers to learn a new programming language, programmers can contribute to the same software project, but write code using any (or several) of the .NET languages (such as C#, Visual Basic .NET, Visual C++ .NET and others) with which they are most competent. In addition to providing language independence, .NET extends program portability by enabling .NET applications to reside on, and communicate across, multiple platforms—thus facilitating the delivery of Web services over the Internet. The .NET platform enables Web-based applications to be distributed to consumer-electronic devices, such as cell phones and personal digital assistants, as well as to desktop computers. The capabilities that Microsoft has incorporated into the .NET platform create a new software-development paradigm that will increase programmer productivity and decrease development time.

New Features in *C# How to Program*

This edition contains many new features and enhancements, including:

- ***Full-Color Presentation.*** This book is now in full color. Full color enables readers to see sample outputs as they would appear on a color monitor. Also, we now syntax color the C# code, similar to the way Visual Studio .NET colors the code in its editor window. Our syntax-coloring conventions are as follows:

  ```
  comments appear in green
  keywords appear in dark blue
  literal values appear in light blue
  text, class, method and variable names appear in black
  errors and ASP .NET directives appear in red
  ```

- ***"Code Washing."*** This is our term for the process we use to format the programs in the book so that they have a carefully commented, open layout. The code appears in full color and is grouped into small, well-documented pieces. This greatly improves code readability—an especially important goal for us, considering that this book contains approximately 23,500 lines of code.
- ***Web Services and ASP .NET.*** Microsoft's .NET strategy embraces the Internet and Web as integral to the software development and deployment processes. Web services—a key technology in this strategy—enables information sharing, commerce and other interactions using standard Internet protocols and technologies, such as Hypertext Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP) and Extensible Markup Language (XML). Web services enable programmers to package application functionality in a form that turns the Web into a library of reusable software components. In Chapter 21, ASP .NET and Web Services, we present a Web service that allows users to make airline seat reservations. In this example, a user accesses a Web page, chooses a seating option and submits the page to the Web server. The page then calls a Web service that checks seat availability. We also present information related to Web services in Appendix P, Crystal Reports® for Visual Studio® .NET, which discusses popular reporting software for database-intensive applications. Crystal Reports, which is integrated into Visual Studio .NET, provides the ability to expose a report as a Web service. The appendix provides introductory information and directs readers to a walkthrough of this process on the Crystal Decisions Web site (**`www.crystaldecisions.com/net`**).
- ***Web Forms, Web Controls and ASP .NET.*** Application developers must be able to create robust, scalable Web-based applications. The .NET platform architecture supports such applications. Microsoft's .NET server-side technology, Active Server Pages (ASP) .NET, allows programmers to build Web documents that respond to client requests. To enable interactive Web pages, server-side programs process information users input into HTML forms. ASP .NET is a significant de-

parture from previous versions of ASP, allowing developers to program Web-based applications using the powerful object-oriented languages of .NET. ASP .NET also provides enhanced visual programming capabilities, similar to those used in building Windows forms for desktop programs. Programmers can create Web pages visually, by dragging and dropping Web controls onto Web forms. Chapter 20, ASP .NET, Web Forms and Web Controls, introduces these powerful technologies.

- ***Object-Oriented Programming.*** Object-oriented programming is the most widely employed technique for developing robust, reusable software, and C# offers enhanced object-oriented programming features. This text offers a rich presentation of object-oriented programming. Chapter 8, Object-Based Programming, introduces how to create classes and objects. These concepts are extended in Chapter 9, Object-Oriented Programming: Inheritance, which discusses how programmers can create new classes that “absorb” the capabilities of existing classes. Chapter 10, Object-Oriented Programming: Polymorphism, familiarizes the reader with the crucial concepts of polymorphism, abstract classes, concrete classes and interfaces, which facilitate powerful manipulations among objects belonging to an inheritance hierarchy.
- ***XML.*** Use of Extensible Markup Language (XML) is exploding in the software-development industry, the e-business and e-commerce communities, and is pervasive throughout the .NET platform. Because XML is a platform-independent technology for describing data and for creating markup languages, XML’s data portability integrates well with C#’s portable applications and services. Chapter 18, Extensible Markup Language (XML), introduces XML. In this chapter, we introduce basic XML markup and discuss the technologies such as DTDs and Schema, which are used to validate XML documents’ contents. We also explain how to programmatically manipulate XML documents using the Document Object Model (DOM™) and how to transform XML documents into other types of documents via Extensible Stylesheet Language Transformations (XSLT).
- ***Multithreading.*** Computers enable us to perform many tasks in parallel (or concurrently), such as printing documents, downloading files from a network and surfing the Web. Multithreading is the technology through which programmers can develop applications that perform concurrent tasks. Historically, a computer has contained a single, expensive processor, which its operating system would share among all applications. Today, processors are becoming so inexpensive that it is possible to build affordable computers that contain many processors that work in parallel—such computers are called multiprocessors. Multithreading is effective on both single-processor and multiprocessor systems. C#’s multithreading capabilities make the platform and its related technologies better prepared to deal with today’s sophisticated multimedia-intensive, database-intensive, network-based, multiprocessor-based distributed applications. Chapter 14, Multithreading, provides a detailed discussion of multithreading.
- ***ADO .NET.*** Databases store vast amounts of information that individuals and organizations must access to conduct business. As an evolution of Microsoft's ActiveX Data Objects (ADO), ADO .NET represents a new approach for building

applications that interact with databases. ADO .NET uses XML and an enhanced object model to provide developers with the tools they need to access and manipulate databases for large-scale, extensible, mission-critical multi-tier applications. Chapter 19, Database, SQL and ADO .NET, details the capabilities of ADO .NET and the Structured Query Language (SQL) to manipulate databases.

- ***Visual Studio .NET Debugger***. Debuggers are programs that help programmers find and correct logic errors in program code. Visual Studio .NET contains a powerful debugging tool that allows programmers to analyze their programs line-by-line as those programs execute. In Appendix D, Visual Studio .NET Debugger, we explain how to use key debugger features, such as setting breakpoints and "watches," stepping into and out of procedures, and examining the procedure call stack.
- ***COM (Component Object Model) Integration***. Prior to the introduction of .NET, many organizations spent tremendous amounts of time and money creating reusable software components called COM components, which include ActiveX® controls and ActiveX DLLs (dynamic link libraries) for Windows applications. In Appendix H, COM Integration, we discuss some of the tools available in Visual Studio .NET for integrating these legacy components into .NET applications. This integration allows programmers to use existing sets of COM-based controls with .NET components.
- ***XML Documentation.*** Documenting program code is crucial for software development, because different programmers often work on an application during the software's lifecycle, which usually includes multiple versions and can span many years. If programmers document software code and methods, other programmers working on the application can learn and understand the logic underlying the code, thus saving time and avoiding misunderstandings. To automate documenting programs, Visual Studio .NET provides an XML tool for C# programmers. Appendix E, XML Documentation, explains how a programmer can insert comments in the code, which produces a separate file providing the code documentation.
- ***Career Opportunities.*** Appendix C, Career Opportunities, introduces career services available on the Internet. We explore online career services from both the employer's and employee's perspectives. We list many Web sites at which you can submit applications, search for jobs and review applicants (if you are interested in hiring someone). We also review services that build recruiting pages directly into e-businesses. One of our reviewers told us that he had used the Internet as a primary tool in a recent job search, and that this appendix would have helped him expand his search dramatically.
- ***Unicode.*** As computer systems evolved worldwide, computer vendors developed numeric representations of character sets and special symbols for the local languages spoken in different countries. In some cases, different representations were developed for the same languages. Such disparate character sets hindered communication among computer systems. C# supports the *Unicode Standard* (maintained by a non-profit organization called the *Unicode Consortium*), which maintains a single character set that specifies unique numeric values for characters and special symbols in most of the world's languages. Appendix G, Unicode, discusses the standard, overviews the Unicode Consortium Web site (**`www.uni-`**

`code.org`) and presents a C# application that displays “Welcome to Unicode!” in several languages.

- ***XHTML.*** The World Wide Web Consortium (W3C) has declared HTML to be a legacy technology that will undergo no further development. HTML is being replaced by the Extensible Hypertext Markup Language (XHTML)—an XML-based technology that is rapidly becoming the standard for describing Web content. We use XHTML in Chapter 18, Extensible Markup Language (XML), and offer an introduction to the technology in Appendix K, Introduction to XHTML: Part 1, and Appendix L, Introduction to XHTML: Part 2. These appendices overview headers, images, lists, image maps and other features of this emerging markup language. (We also present a treatment of HTML in Appendices I and J, because ASP .NET, used in Chapters 20 and 21, generates HTML content).
- ***Accessibility.*** Although the World Wide Web has become an important part of many people’s lives, the medium currently presents many challenges to people with disabilities. Individuals with hearing and visual impairments, in particular, have difficulty accessing multimedia-rich Web sites. In an attempt to improve this situation, the World Wide Web Consortium (W3C) launched the Web Accessibility Initiative (WAI), which provides guidelines for making Web sites accessible to people with disabilities. Chapter 24, Accessibility, describes these guidelines and highlights various products and services designed to improve the Web-browsing experiences of individuals with disabilities. For example, the chapter introduces VoiceXML and CallXML—two XML-based technologies for increasing the accessibility of Web-based content for people with visual impairments.
- ***Bit Manipulation.*** Computers work with data in the form of binary digits, or bits, which can assume the values 1 or 0. Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from 1 to 0 or from 0 to 1). Operating systems, test-equipment, networking software and many other kinds of software require that programs communicate “directly with the hardware” by using bit manipulation. Appendix O, Bit Manipulation, overviews the bit manipulation capabilities that the .NET Framework provides.

Some Notes to Instructors

Students Enjoy Learning a Leading-Edge Language

Dr. Harvey M. Deitel taught introductory programming courses in universities for 20 years with an emphasis on developing clearly written, well-designed programs. Much of what is taught in such courses represents the basic principles of programming, concentrating on the effective use of data types, control structures, arrays and functions. Our experience has been that students handle the material in this book in about the same way that they handle other introductory and intermediate programming courses. There is one noticeable difference, though: Students are highly motivated by the fact that they are learning a leading-edge language, C#, and a leading-edge programming paradigm (object-oriented programming) that will be immediately useful to them as they enter the business world. This increases their enthusiasm for the material—which is essential when you consider that there is much more to learn in a C# course now that students must master both the base language and substantial class libraries as well. Although C# is a new language that may require programmers to revamp their skills, programmers will be motivated to do so because of the powerful range of capabilities that Microsoft is offering in its .NET initiative.

A World of Object Orientation

In the late 1990s, universities were still emphasizing procedural programming. The leading-edge courses were using object-oriented C++, but these courses generally mixed a substantial amount of procedural programming with object-oriented programming—something that C++ lets programmers do. Many instructors now are emphasizing a pure object-oriented programming approach. This book—the first edition of *C# How to Program* and the second text in our .NET series—takes a predominantly object-oriented approach because of the object orientation provided in C#.

Focus of the Book

Our goal was clear: Produce a C# textbook for introductory university-level courses in computer programming aimed at students with little or no programming experience, yet offer the depth and the rigorous treatment of theory and practice demanded by both professionals and students in traditional, upper-level programming courses. To meet these objectives, we produced a comprehensive book that patiently teaches the principles of computer programming and of the C# language, including control structures, object-oriented programming, C# class libraries, graphical-user-interface concepts, event-driven programming and more. After mastering the material in this book, students will be well-prepared to program in C# and to employ the capabilities of the .NET platform.

Multimedia-Intensive Communications

People want to communicate. Sure, they have been communicating since the dawn of civilization, but the potential for information exchange has increased dramatically with the evolution of various technologies. Until recently, even computer communications were limited mostly to digits, alphabetic characters and special characters. The current wave of communication technology involves the distribution of multimedia—people enjoy using applications that transmit color pictures, animations, voices, audio clips and even full-motion color video over the Internet. At some point, we will insist on three-dimensional, moving-image transmission.

There have been predictions that the Internet will eventually replace radio and television as we know them today. Similarly, it is not hard to imagine newspapers, magazines and books delivered to “the palm of your hand” (or even to special eyeglasses) via wireless communications. Many newspapers and magazines already offer Web-based versions, and some of these services have spread to the wireless world. When cellular phones were first introduced, they were large and cumbersome. Today, they are small devices that fit in our pockets, and many are Internet-enabled. Given the current rate of advancement, wireless technology soon could offer enhanced streaming-video and graphics-packed services, such as video conference calls and high-power, multi-player video games.

Teaching Approach

C# How to Program contains a rich collection of examples, exercises and projects drawn from many fields and designed to provide students with a chance to solve interesting, real-world problems. The code examples in this text have been tested on Windows 2000 and

Windows XP. The book concentrates on the principles of good software engineering, and stresses program clarity. We are educators who teach edge-of-the-practice topics in industry classrooms worldwide. We avoid arcane terminology and syntax specifications in favor of teaching by example. The text emphasizes good pedagogy.[1]

LIVE-CODE™ Teaching Approach

C# How to Program is loaded with numerous LIVE-CODE™ examples. This style exemplifies the way we teach and write about programming and is the focus of our multimedia *Cyber Classrooms* and Web-based training courses. Each new concept is presented in the context of a complete, working example that is immediately followed by one or more windows showing the program's input/output dialog. We call this method of teaching and writing the ***LIVE-CODE™ Approach***. *We use programming languages to teach programming languages.* Reading the examples in the text is much like entering and running them on a computer.

World Wide Web Access

All of the examples for *C# How to Program* (and our other publications) are available on the Internet as downloads from the following Web sites:

```
www.deitel.com
www.prenhall.com/deitel
```

Registration is quick and easy and these downloads are free. We suggest downloading all the examples, then running each program as you read the corresponding text. Make changes to the examples and immediately see the effects of those changes—a great way to learn programming. Each set of instructions assumes that the user is running Windows 2000 or Windows XP and is using Microsoft's Internet Information Services (IIS). Additional setup instructions for Web servers and other software can be found at our Web sites along with the examples. [*Note:* This is copyrighted material. Feel free to use it as you study, but you may not republish any portion of it in any form without explicit permission from Prentice Hall and the authors.]

Visual Studio .NET, which includes C#, can be purchased and downloaded from Microsoft. Three different versions of Visual Studio .NET are available—Enterprise, Professional and Academic. Visit **`developerstore.com/devstore/`** for more details and to order. If you are a member of the Microsoft Developer Network, visit **`msdn.microsoft.com/default.asp`**.

Objectives

Each chapter begins with objectives that inform students of what to expect and give them an opportunity, after reading the chapter, to determine whether they have met the intended goals. The objectives serve as confidence builders and as a source of positive reinforcement.

Quotations

The chapter objectives are followed by sets of quotations. Some are humorous, some are philosophical and some offer interesting insights. We have found that students enjoy relating the quotations to the chapter material. Many of the quotations are worth a “second look” *after* you read each chapter.

---

[1] We use fonts to distinguish between Visual Studio .NET's Integrated Development Environment (IDE) features (such as menu names and menu items) and other elements that appear in the IDE. Our convention is to emphasize IDE features in a sans-serif bold Helvetica font (e.g., **Project** menu) and to emphasize program text in a serif bold Courier font (e.g., **`bool x = true;`**).

Outline

The chapter outline enables students to approach the material in top-down fashion. Along with the chapter objectives, the outline helps students anticipate future topics and set a comfortable and effective learning pace.

Approximately 23,500 Lines of Code in 204 Example Programs (with Program Outputs)

We present C# features in the context of complete, working C# programs. The programs range in size from just a few lines of code to substantial examples containing several hundred lines of code. All examples are available on the CD that accompanies the book or as downloads from our Web site, **`www.deitel.com`**.

607 Illustrations/Figures

An abundance of charts, line drawings and program outputs is included. The discussion of control structures, for example, features carefully drawn flowcharts. [*Note:* We do not teach flowcharting as a program-development tool, but we do use a brief, flowchart-oriented presentation to explain the precise operation of each C# control structure.]

509 Programming Tips

We have included programming tips to help students focus on important aspects of program development. We highlight hundreds of these tips in the form of *Good Programming Practices*, *Common Programming Errors*, *Testing and Debugging Tips*, *Performance Tips*, *Portability Tips*, *Software Engineering Observations* and *Look-and-Feel Observations*. These tips and practices represent the best the authors have gleaned from a combined seven decades of programming and teaching experience. One of our students—a mathematics major—told us that she feels this approach is like the highlighting of axioms, theorems and corollaries in mathematics books; it provides a foundation on which to build good software.

91 Good Programming Practices

Good Programming Practices *are tips that call attention to techniques that will help students produce better programs. When we teach introductory courses to nonprogrammers, we state that the “buzzword” for each course is “clarity,” and we tell the students that we will highlight (in these Good Programming Practices) techniques for writing programs that are clearer, more understandable and more maintainable.*

165 Common Programming Errors

Students learning a language—especially in their first programming course—tend to make certain kinds of errors frequently. Pointing out these Common Programming Errors *reduces the likelihood that students will make the same mistakes. It also shortens long lines outside instructors’ offices during office hours!*

44 Testing and Debugging Tips

When we first designed this “tip type,” we thought the tips would contain suggestions strictly for exposing bugs and removing them from programs. In fact, many of the tips describe aspects of C# that prevent “bugs” from getting into programs in the first place, thus simplifying the testing and debugging process.

57 Performance Tips

In our experience, teaching students to write clear and understandable programs is by far the most important goal for a first programming course. But students want to write programs that run the fastest, use the least memory, require the smallest number of keystrokes or dazzle in other ways. Students really care about performance and they want to know what they can do to "turbo charge" their programs. We have included 57 Performance Tips *that highlight opportunities for improving program performance—making programs run faster or minimizing the amount of memory that they occupy.*

16 Portability Tips

We include Portability Tips *to help students write portable code and to provide insights on how C# achieves its high degree of portability.*

115 Software Engineering Observations

The object-oriented programming paradigm necessitates a complete rethinking of the way we build software systems. C# is an effective language for achieving good software engineering. The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems. Much of what the student learns here will be useful in upper-level courses and in industry as the student begins to work with large, complex real-world systems.*

21 Look-and-Feel Observations

We provide Look-and-Feel Observations *to highlight graphical-user-interface conventions. These observations help students design attractive, user-friendly graphical user interfaces that conform to industry norms.*

Summary (1277 Summary bullets)

Each chapter ends with additional pedagogical devices. We present a thorough, bullet-list-style summary of the chapter. On average, there are 39 summary bullets per chapter. This helps the students review and reinforce key concepts.

Terminology (2932 Terms)

We include an alphabetized list of the important terms defined in the chapter in a *Terminology* section. Again, this serves as further reinforcement. On average, there are 89 terms per chapter. Each term also appears in the index, so the student can locate terms and definitions quickly.

693 Self-Review Exercises and Answers (Count Includes Separate Parts)

Extensive self-review exercises and answers are included for self-study. These questions and answers give the student a chance to build confidence with the material and prepare for the regular exercises. Students should be encouraged to attempt all the self-review exercises and check their answers.

367 Exercises (Solutions in Instructor's Manual; Count Includes Separate Parts)

Each chapter concludes with a substantial set of exercises that involve simple recall of important terminology and concepts; writing individual C# statements; writing small portions of C# methods and classes; writing complete C# methods, classes and applications; and writing major projects. These exercises cover a wide variety of topics, enabling instructors to tailor their courses to the unique needs of their audiences and to vary course assignments each semester. Instructors can use the exercises to form homework assignments, short quizzes and major examinations. The solutions for the exercises are included in the *Instructor's Manual* and on the disks *available only to instructors* through their Prentice-Hall representatives. **[NOTE: Please do not write to us requesting the instructor's manual. Distribution of this publication is strictly limited to college professors teaching from the book. Instructors may obtain the solutions manual from their regular Prentice Hall representatives. We regret that we cannot provide the solutions to professionals.]** Solutions to approximately half the exercises are included on the *C# Multimedia Cyber Classroom* CD-ROM (available in April 2002 at **`www.InformIT.com/cyberclassrooms`**; also see the last few pages of this book or visit **`www.deitel.com`** for ordering instructions). Also available in April 2002 is the boxed product, *The Complete C# Training Course*, which includes both our textbook, *C# How to Program* and the *C# Multimedia Cyber Classroom*. All of our *Complete Training Course* products are available at bookstores and online booksellers, including **`www.InformIT.com`**.

Approximately 5,420 Index Entries (with approximately 6,450 Page References)

We have included an extensive Index at the back of the book. Using this resource, students can search for any term or concept by keyword. The Index is especially useful to practicing programmers who use the book as a reference. Each of the 2932 terms in the Terminology sections appears in the Index (along with many more index items from each chapter). Students can use the index in conjunction with the Terminology sections to ensure that they have covered the key material in each chapter.

"Double Indexing" of All C# LIVE-CODE™ Examples

C# How to Program has 204 LIVE-CODE™ examples, which we have "double indexed." For every C# source-code program in the book, we took the file name with the **`.cs`** extension, such as **`ChessGame.cs`**, and indexed it both alphabetically (in this case, under "C") and as a subindex item under "Examples." This makes it easier to find examples using particular features.

C# Multimedia Cyber Classroom and The Complete C# Training Course,

We have prepared an interactive, CD-ROM-based, software version of *C# How to Program*, called the *C# Multimedia Cyber Classroom.* This resource is loaded with e-Learning features that are ideal for both learning and reference. The *Cyber Classroom* is packaged with the textbook at a discount in *The Complete C# Training Course*. If you already have the book and would like to purchase the *C# Multimedia Cyber Classroom* separately, please visit **`www.InformIT.com/cyberclassrooms`**. The ISBN number for the *C# Multimedia Cyber Classroom* is 0-13-064587-7. All Deitel™ *Cyber Classrooms* are available in CD-ROM and Web-based training formats.

The CD provides an introduction in which the authors overview the *Cyber Classroom*'s features. The textbook's 204 LIVE-CODE™ example C# programs truly "come alive" in the *Cyber Classroom*. If you are viewing a program and want to execute it, you simply click the lightning-bolt icon, and the program will run. You immediately will see—and hear, when working with audio-based multimedia programs—the program's outputs. If you want to modify a program and see the effects of your changes, simply click the floppy-disk icon that causes the source code to be “lifted off” the CD and “dropped into” one of your own directories so you can edit the text, recompile the program and try out your new version. Click the audio icon, and one of the authors will discuss the program and “walk you through” the code.

The *Cyber Classroom* also provides navigational aids, including extensive hyperlinking. The *Cyber Classroom* is browser based, so it remembers sections that you have visited recently and allows you to move forward or backward among these sections. The thousands of index entries are hyperlinked to their text occurrences. Furthermore, when you key in a term using the “find” feature, the *Cyber Classroom* will locate occurrences of that term throughout the text. The Table of Contents entries are “hot,” so clicking a chapter name takes you immediately to that chapter.

Students like the fact that solutions to approximately half the exercises in the book are included with the *Cyber Classroom*. Studying and running these extra programs is a great way for students to enhance their learning experience.

Students and professional users of our *Cyber Classrooms* tell us that they like the interactivity and that the *Cyber Classroom* is an effective reference due to its extensive hyperlinking and other navigational features. We received an e-mail from a person who said that he lives “in the boonies” and cannot take a live course at a university, so the *Cyber Classroom* provided an ideal solution to his educational needs.

Professors tell us that their students enjoy using the *Cyber Classroom* and spend more time on the courses and master more of the material than in textbook-only courses. For a complete list of the available and forthcoming *Cyber Classrooms* and *Complete Training Courses*, see the *Deitel™ Series* page at the beginning of this book, the product listing and ordering information at the end of this book or visit `www.deitel.com`, `www.prenhall.com/deitel` and `www.InformIT.com/deitel`.

Deitel e-Learning Initiatives

e-Books and Support for Wireless Devices

Wireless devices will play an enormous role in the future of the Internet. Given recent bandwidth enhancements and the emergence of 2.5 and 3G technologies, it is projected that, within two years, more people will access the Internet through wireless devices than through desktop computers. Deitel & Associates, Inc., is committed to wireless accessibility and has recently published *Wireless Internet & Mobile Business How to Program*. To fulfill the needs of a wide range of customers, we currently are developing our content both in traditional print formats and in newly developed electronic formats, such as e-books so that students and professors can access content virtually anytime, anywhere. Visit `www.deitel.com` for periodic updates on this initiative.

e-Matter

Deitel & Associates, Inc., is partnering with Prentice Hall’s parent company, Pearson PLC, and its information technology Web site, `InformIT.com`, to launch the Deitel e-Matter series at `www.InformIT.com/deitel`. This series will provide professors, students and professionals with an additional source of information on specific programming topics. e-Matter consists of stand-alone sections taken from published texts, forthcoming texts or pieces written during the Deitel research-and-development process. Developing e-Matter based on pre-publication books allows us to offer significant amounts of the material to early adopters for use in courses. Some possible C# e-Matter titles we are considering include *Object-Based Programming and Object-Oriented Programming in C#*; *Graphical User Interface Programming in C#*; *Multithreading in C#*; *ASP .NET and Web Forms: A C# View;* and *ASP .NET and Web Services: A C# View*.

Course Management Systems: WebCT, Blackboard, and CourseCompass
We are working with Prentice Hall to integrate our *How to Program Series* courseware into three Course Management Systems: WebCT, Blackboard™ and CourseCompass. These Course Management Systems enable instructors to create, manage and use sophisticated Web-based educational programs. Course Management System features include course customization (such as posting contact information, policies, syllabi, announcements, assignments, grades, performance evaluations and progress tracking), class and student management tools, a gradebook, reporting tools, communication tools (such as chat rooms), a whiteboard, document sharing, bulletin boards and more. Instructors can use these products to communicate with their students, create online quizzes and tests from questions directly linked to the text and automatically grade and track test results. For more information about these upcoming products, visit **`www.deitel.com/whatsnew.html`**. For demonstrations of existing WebCT, Blackboard and CourseCompass courses, visit **`cms.pren_hall.com/WebCT`**, **`cms.prenhall.com/Blackboard`** and **`cms.prenhall.com/CourseCompass`**, respectively.

Deitel and InformIT Newsletters

Deitel Column in the InformIT Newsletters
Deitel & Associates, Inc., contributes a weekly column to the popular *InformIT* newsletter, currently subscribed to by more than 800,000 IT professionals worldwide. For opt-in registration, visit **`www.InformIT.com`**.

Deitel Newsletter
Our own free, opt-in newsletter includes commentary on industry trends and developments, links to articles and resources from our published books and upcoming publications, information on future publications, product-release schedules and more. For opt-in registration, visit **`www.deitel.com`**.

The Deitel .NET Series

Deitel & Associates, Inc., is making a major commitment to .NET programming through the launch of our .NET Series. *C# .NET How to Program* and *Visual Basic .NET How to Program, Second Edition* are the first books in this new series. We intend to follow these books with *Advanced C# How to Program* and *Advanced Visual Basic .NET How to Program*, which will be published in December 2002. We also plan to publish *Visual C++ .NET How to Program* in July 2002, followed by *Advanced Visual C++ .NET How to Program* in July 2003.

Advanced C# How to Program

C# How to Program covers introductory through intermediate-level C# programming topics, as well as core programming fundamentals. By contrast, our upcoming textbook *Ad-*

vanced C# How to Program will be geared toward experienced C# developers. This new book will cover enterprise-level programming topics, including: Creating multi-tier, database intensive ASP .NET applications using ADO .NET and XML; constructing custom Windows controls; developing custom Web controls; and building Windows services. The book also will include more in-depth explanations of object-oriented programming (with the UML), ADO .NET, XML Web services, wireless programming and security. *Advanced C# How to Program* will be published in December 2002.

Acknowledgments

One of the great pleasures of writing a textbook is acknowledging the efforts of many people whose names may not appear on the cover, but whose hard work, cooperation, friendship and understanding were crucial to the production of the book.

Many other people at Deitel & Associates, Inc., devoted long hours to this project.

- Sean E. Santry, a graduate of Boston College with degrees in Computer Science and Philosophy, Director of Software Development at Deitel & Associates, Inc., and co-author of *Advanced Java 2 Platform How to Program*, contributed to Chapters 1–10, 12–13 and 18–23.
- Matthew R. Kowalewski, a graduate of Bentley College with a degree in Accounting Informations Systems, is the Director of Wireless Development at Deitel & Associates, Inc. He contributed to Chapters 19–20, Appendices B, F, I–N, P and edited the Index.
- Jonathan Gadzik, a graduate of the Columbia University School of Engineering and Applied Science with a major in Computer Science, co-authored Chapter 17 and contributed to Chapters 9, 22 and Appendices D and E.
- Kyle Lomelí, a graduate of Oberlin College with a degree in Computer Science and a minor in East Asian Studies, contributed to Chapters 11, 14–15, 19 and 24.
- Lauren Trees, a graduate if Brown University in English, edited the entire manuscript for smoothness, clarity and effectiveness of presentation; she also co-authored the Preface, Chapter 1 and Appendix P.
- Rashmi Jayaprakash, a graduate of Boston University with a major in Computer Science, co-authored Chapter 24 and Appendix G.
- Laura Treibick, a graduate of the University of Colorado at Boulder with a degree in Photography and Multimedia, is Director of Multimedia at Deitel & Associates, Inc. She contributed to Chapter 16 and enhanced many of the graphics throughout the text.
- Betsy DuWaldt, a graduate of Metropolitan State College of Denver with a major in Technical Communications (Writing and Editing emphasis) and a minor in Computer Information Systems, is Editorial Director at Deitel & Associates, Inc. She co-authored the Preface, Chapter 1 and Appendix P and managed the permissions process for the book.
- Barbara Deitel applied the copy edits to the manuscript. She did this in parallel with handling her extensive financial and administrative responsibilities at Deitel

& Associates, Inc., which include serving as Chief Financial Officer. [Everyone at the company works on book content.]

- Abbey Deitel, a graduate of Carnegie Mellon University's Industrial Management Program and President of Deitel & Associates, Inc., recruited 40 additional full-time employees and interns during 2001. She also leased, equipped and furnished our second building to create the work environment from which *C# How to Program* and our other year 2001 publications were produced. She suggested the title for the *How to Program* series, and edited this preface and several of the book's chapters.

We would also like to thank the participants in the Deitel & Associates, Inc., College Internship Program.[2]

- Jeffrey Hamm, a sophomore at Northeastern University in Computer Science, co-authored Chapters 16, 18, 20–21 and Appendices D and H.
- Kalid Azad, a sophomore at Princeton University in Computer Science, contributed to Chapters 1, 2, 12–13, 16 and Appendix D. He created PowerPoint-slide ancillaries for Chapters 1–7 and researched Visual Studio .NET and Microsoft's .NET initiative.
- Christopher Cassa, a junior at MIT in Computer Science, contributed to Chapters 3–7 and 18.
- David Tuttle, a senior at Harvard in Computer Science, contributed to Chapters 8, 18–19 and 24 and coded examples for Chapters 3–6, 7, 11,16–17,19, 23 and 26.
- Ori Schwartz, a sophomore at Boston University in Computer Science, produced solutions for all the chapters and contributed to Chapter 16.
- Thiago Lucas da Silva, a sophomore at Northeastern University in Computer Science, tested all the programming examples through the various beta releases and release candidates of Visual Studio .NET.
- Matthew Rubino, a sophomore at Northeastern University in Computer Science, created ancillary materials for the entire book.
- Elizabeth Rockett, a senior in English at Princeton University, edited 1-3, 7–8, 14, 17 and 19-24.
- Barbara Strauss, a senior in English at Brandeis University, edited Chapters 1–6, 9–13 and 18–24.
- Christina Carney, a senior in Psychology and Business at Framingham State College, helped with the Preface.

---

[2] The *Deitel & Associates, Inc. College Internship Program* offers a limited number of salaried positions to Boston-area college students majoring in Computer Science, Information Technology, Marketing, Management and English. Students work at our corporate headquarters in Sudbury, Massachusetts full-time in the summers and (for those attending college in the Boston area) part-time during the academic year. We also offer full-time internship positions for students interested in taking a semester off from school to gain industry experience. Regular full-time positions are available to college graduates. For more information about this competitive program, please contact Abbey Deitel at **`deitel@deitel.com`** and visit our Web site, **`www.deitel.com`**.

- Reshma Khilnani, a junior in Computer Science and Mathematics at Massachusetts Institute of Technology, contributed to Chapter 18 and Appendix E.
- Brian Foster, a sophomore at Northeastern University in Computer Science, helped with the Preface and Bibliography.
- Mike Preshman, a sophomore at Northeastern University with a major in Computer Science and minors in Electrical Engineering and Math, helped with the Bibliography.

We are fortunate to have been able to work on this project with the talented and dedicated team of publishing professionals at Prentice Hall. We especially appreciate the extraordinary efforts of our Computer Science editor, Petra Recter and her boss—our mentor in publishing—Marcia Horton, Editorial Director of Prentice-Hall's Engineering and Computer Science Division. Camille Trentacoste and her boss Vince O'Brien did a marvelous job managing the production of the book. Sarah Burrows handled editorial responsibilities on the book's extensive ancillary package.

The *C# Multimedia Cyber Classroom* was developed in parallel with *C# How to Program*. We sincerely appreciate the "new media" insight, savvy and technical expertise of our electronic-media editors, Mark Taub and Karen McLean. They and project manager Mike Ruel did a wonderful job bringing the *C# Multimedia Cyber Classroom* and *The Complete C# Training Course* to publication.

We owe special thanks to the creativity of Tamara Newnam (**`smart_art@earthlink.net`**), who produced the art work for our programming-tip icons and for the cover. She created the delightful creature who shares with you the book's programming tips. Barbara Deitel and Abbey Deitel contributed the bugs' names for the front cover.

During the development of this manuscript, we were fortunate to have had two universities—the Massachusetts Institute of Technology and Yale University—beta-test the book in the Fall 2001 semester. MIT Professor John Williams used the text to teach the graduate-level class, *Web System Architecting—Part I: Programming Clients and Web Services Using C# and .NET*, for the Off-Campus Advanced Study Program. Chris Cassa, a summer 2001 intern at Deitel & Associates, Inc., was the teaching fellow for the class. Yale Professor Paul Hudak used the manuscript for an *Introduction to Programming* class, which taught object-oriented programming languages. We would like to thank Professor Williams, Professor Hudak and Chris for their contributions. The feedback we received was crucial to fine-tuning this text.

We wish to acknowledge the efforts of our first- and second-round reviewers and to thank Crissy Statuto and Jennifer Cappello of Prentice Hall, who recruited the reviewers and managed the review process. Adhering to a tight time schedule, these reviewers scrutinized the text and the programs, providing countless suggestions for improving the accuracy and completeness of the presentation. It is a privilege to have the guidance of such talented and busy professionals.

***C# How to Program reviewers*:**
Hussein Abuthuraya (Microsoft)
Lars Bergstrom (Microsoft)
Indira Dhingra (Microsoft)
Eric Gunnerson (Microsoft)
Peter Hallam (Microsoft)

Habib Hegdarian (Microsoft)
Anson Horton (Microsoft)
Latha Lakshminaray (Microsoft)
Kerry Loynd (Microsoft)
Tom McDade (Microsoft)
Syed Mehdi (Microsoft)
Cosmin Radu (Microsoft)
Ratta Rakshminarayana (Microsoft)
Imtiaz Syed (Microsoft)
Ed Thornburg (Microsoft)
Richard Van Fossen (Microsoft)
Rishabh Agarwal (Delteq Systems Pte. Ltd.)
José Antonio González Seco (Sadiel S.A.)
Paul Bohman (WebAIM)
Alex Bondarev (SureFire Commerce, Inc.)
Ron Braithwaite (Nutriware)
Filip Bulovic (Objectronics PTY Ltd.)
Mark Burhop (University of Cincinnati)
Carl Burnham (Southpoint)
Matt Butler (Oakscape Inc.)
Andrew Chau (Rich Solutions, Inc.)
Dharmesh Chauhan (Microsoft Consultant, Singapore)
Shyam Chebrolu (SAIC Broadway & Seymour Group)
Kunal Cheda (DotNetExtreme.com)
Edmund Chou (MIT Student, www.devhood.com project, Microsoft Intern)
James Chegwidden (Tarrant County College)
Vijay Cinnakonda (University of Toledo)
Michael Colynuck (Sierra Systems)
Jay Cook (Canon Information Systems)
Jeff Cowan (Magenic Technologies)
Robert Dombroski (AccessOnTime)
Shaun Eagan ((Eagan Consulting)
Brian Erwin (Extreme Logic)
Hamilton Fong (Montag & Caldwell, Inc.)
Gnanavel Gnana Arun Ganesh (Arun Microsystems)
Sam Gentile (Consultant)
Sam Gill (San Francisco State University)
John Godel (TJX)
Dave Haglin (Minnesota State University in Mankato)
Jeff Isom (WebAIM)
Rex Jaeschke (Consultant)
Amit Kalani (MobiCast)
Priti Kalani (Consultant)
Bryan Keller (csharphelp.com)
Patrick Lam (EdgeNet Communications)
Yi-Fung Lin (MIT Student, www.devhood.com project, Microsoft Intern)

Maxim Loukianov (SoloMio Corporation)
Guarav Mantro (EDS PLM Solutions)
Jaimon Mathew (Osprey Software Technology)
Robert Meagher (Compuware NuMega Lab)
Arun Nair (iSpan Technologies)
Saurabh Nandu (Mastercsharp.com)
Simon North (Synopsys)
Jibin Pan (csharpcorner.com)
Graham Parker (VBUG)
Bryan Plaster (Valtech)
Chris Rausch (Sheridan Press)
Debbie Reid (Santa Fe Community College)
Bryn Rhodes (Softwise, Inc.)
Craig Schofding (C.A.S. Training)
Rahul Sharma (Maxutil Software)
Devan Shepherd (XMaLpha Technologies)
David Talbot (Reallinx, Inc.)
Satish Talim (Pune-Csharp)
Pavel Tsekov (Consultant)
John Varghese (UBS Warburg)
Peter Weng (MIT Student, www.devhood.com project, Microsoft Intern)
Jesse Wilkins (Metalinear Media)
Warren Wiltsie (Fairleigh Dickinson University/Seton Hall University)
Phil Wright (Crownwood Consulting Ltd.)
Norimasa Yoshida (MIT Graduate Student)

We would sincerely appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to:

`deitel@deitel.com`

We will respond promptly.

Well, that's it for now. Welcome to the exciting world of C# programming. We hope you enjoy this look at leading-edge computer applications. Good luck!

Dr. Harvey M. Deitel
Paul J. Deitel
Tem R. Nieto
Cheryl H. Yaeger
Marina Zlatkina
Jeff Listfield

About the Authors

Dr. Harvey M. Deitel, CEO and Chairman of Deitel & Associates, Inc., has 40 years experience in the computing field, including extensive industry and academic experience. Dr. Deitel earned B.S. and M.S. degrees from the Massachusetts Institute of Technology and a Ph.D. from Boston University. He worked on the pioneering virtual-memory operating-systems projects at IBM and MIT that developed techniques now widely implemented in systems such as UNIX, Linux and Windows NT. He has 20 years of college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., with his son, Paul J. Deitel. He is the author or co-author of several dozen books and multimedia packages and is writing many more. With translations published in Japanese, Russian, Spanish, Traditional Chinese, Simplified Chinese, Korean, French, Polish, Italian and Portuguese, Dr. Deitel's texts have earned international recognition. Dr. Deitel has delivered professional seminars to major corporations and to government organizations and various branches of the military.

Paul J. Deitel, Executive Vice President and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of the Massachusetts Institute of Technology's Sloan School of Management, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered Java, C, C++, Internet and World Wide Web courses to industry clients including Compaq, Sun Microsystems, White Sands Missile Range, Rogue Wave Software, Boeing, Dell, Stratus, Fidelity, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Lucent Technologies, Adra Systems, Entergy, CableData Systems, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, IBM and many other organizations. He has lectured on C++ and Java for the Boston Chapter of the Association for Computing Machinery and has taught satellite-based Java courses through a cooperative venture of Deitel & Associates, Inc., Prentice Hall and the Technology Education Network. He and his father, Dr. Harvey M. Deitel, are the world's best-selling Computer Science textbook authors.

Tem R. Nieto, Director of Product Development of Deitel & Associates, Inc., is a graduate of the Massachusetts Institute of Technology, where he studied engineering and computing. Through Deitel & Associates, Inc., he has delivered courses for industry clients including Sun Microsystems, Compaq, EMC, Stratus, Fidelity, NASDAQ, Art Technology, Progress Software, Toys "R" Us, Operational Support Facility of the National Oceanographic and Atmospheric Administration, Jet Propulsion Laboratory, Nynex, Motorola, Federal Reserve Bank of Chicago, Banyan, Schlumberger, University of Notre Dame, NASA, various military installations and many others. He has co-authored numerous books and multimedia packages with the Deitels and has contributed to virtually every Deitel & Associates, Inc., publication.

Cheryl H. Yaeger, Director of Microsoft Software Publications with Deitel & Associates, Inc., graduated from Boston University in 3 years with a bachelor's degree in Computer Science. Other Deitel publications she has contributed to include *Perl How to Program, Wireless Internet & Mobile Business How to Program* and *Internet and World Wide Web How to Program, Second Edition*. Cheryl is increasingly interested in Microsoft's .NET strategy and in learning how Microsoft's .NET initiative will develop in the coming year.

Marina Zlatkina graduated from Brandeis University in three years with degrees in Computer Science and Mathematics and is pursuing a Master's degree in Computer Science at Brandeis. During her Brandeis career, she has conducted research in databases and has been a teaching assistant. She has also contributed to the Deitel & Associates, Inc. publication, *e-Business & e-Commerce for Managers*.

Jeff Listfield is a senior at Harvard College in Computer Science. His coursework includes classes in computer graphics, networks and computational theory and he has programming experience in C, C++, Java, Perl and Lisp. Jeff also contributed to the Deitel & Associates, Inc., publication *Perl How to Program*.

About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized corporate training and content-creation organization specializing in Internet/World Wide Web software technology, e-business/e-commerce software technology, object technology and computer programming languages education. The company provides courses on Internet and World Wide Web/ programming, wireless Internet programming, object technology, and major programming languages and platforms, such as Visual Basic .NET, C#, Java, advanced Java, C, C++, XML, Perl, Python and more. The founders of Deitel & Associates, Inc., are Dr. Harvey M. Deitel and Paul J. Deitel. The company's clients include many of the world's largest computer companies, government agencies, branches of the military and business organizations. Through its 25-year publishing partnership with Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks, professional books, interactive CD-ROM-based multimedia *Cyber Classrooms*, *Complete Training Courses*, e-books, e-matter, Web-based training courses and course management systems e-content. Deitel & Associates, Inc., and the authors can be reached via e-mail at:

`deitel@deitel.com`

To learn more about Deitel & Associates, Inc., its publications and its worldwide corporate on-site curriculum, see the last few pages of this book or visit:

`www.deitel.com`

Individuals wishing to purchase Deitel books, *Cyber Classrooms*, *Complete Training Courses* and Web-based training courses can do so through bookstores, online booksellers and:

`www.deitel.com`
`www.prenhall.com/deitel`
`www.InformIT.com/deitel`
`www.InformIT.com/cyberclassrooms`

Bulk orders by corporations and academic institutions should be placed directly with Prentice Hall. See the last few pages of this book for worldwide ordering details.

The World Wide Web Consortium (W3C)

Deitel & Associates, Inc., is a member of the *World Wide Web Consortium (W3C)*. The W3C was founded in 1994 "to develop common protocols for the evolution of the World Wide Web." As a W3C member, Deitel & Associates, Inc., holds a seat on the W3C Advisory Committee (the company's representative is our Chief Technology Officer, Paul Deitel). Advisory Committee members help provide "strategic direction" to the W3C through meetings held around the world. Member organizations also help develop standards recommendations for Web technologies (such as XHTML, XML and many others) through participation in W3C activities and groups. Membership in the W3C is intended for companies and large organizations. To obtain information on becoming a member of the W3C visit **`www.w3.org/Consortium/Prospectus/Joining`**.

1

Introduction to Computers, the Internet, the Web and C#

Objectives

- To understand basic computer concepts.
- To learn about various programming languages.
- To become familiar with the history of the C# programming language.
- To understand the Microsoft® .NET initiative.
- To preview the remaining chapters of the book.

Things are always at their best in their beginning.
Blaise Pascal

High thoughts must have high language.
Aristophanes

Our life is frittered away by detail…Simplify, simplify.
Henry David Thoreau

Before beginning, plan carefully….
Marcus Tullius Cicero

Look with favor upon a bold beginning.
Virgil

I think I'm beginning to learn something about it.
Auguste Renoir

Outline

1.1 Introduction
1.2 What Is a Computer?
1.3 Computer Organization
1.4 Evolution of Operating Systems
1.5 Personal Computing, Distributed Computing and Client/Server Computing
1.6 Machine Languages, Assembly Languages and High-level Languages
1.8 C#
1.7 C, C++, Visual Basic .NET and Java™
1.9 Other High-level Languages
1.10 Structured Programming
1.11 Key Software Trend: Object Technology
1.12 Hardware Trends
1.13 History of the Internet and World Wide Web
1.14 World Wide Web Consortium (W3C)
1.15 Extensible Markup Language (XML)
1.16 Introduction to Microsoft .NET
1.17 .NET Framework and the Common Language Runtime
1.18 Tour of the Book
1.19 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

1.1 Introduction

Welcome to C#! In creating this book, we have worked hard to provide students with the most accurate and complete information regarding the C# language, and the .NET platform. The book is designed to be appropriate for readers at all levels, from practicing programmers to individuals with little or no programming experience. We hope that working with this text will be an informative, entertaining and challenging learning experience for you.

How can one book appeal to both novices and skilled programmers? The core of this book emphasizes the achievement of program clarity through proven techniques of *structured programming*, *object-based programming*, *object-oriented programming (OOP)* and *event-driven programming*. Nonprogrammers learn basic skills that underlie good programming; experienced developers receive a rigorous explanation of the language and may improve their programming styles. Perhaps most importantly, the book presents hundreds of complete, working C# programs and depicts their outputs. We call this the *LIVE-CODE™ approach*. All of the book's examples are available on the CD-ROM that accompanies this book and on our Web site, **`www.deitel.com`**.

Computer use is increasing in almost every field of endeavor. In an era of steadily rising costs, computing costs have decreased dramatically because of rapid developments in both hardware and software technology. Computers that filled large rooms and cost millions of dollars just two decades ago now can be inscribed on the surfaces of silicon chips smaller than a fingernail, costing perhaps a few dollars each. Silicon is one of the most abundant materials on earth—it is an ingredient in common sand. Silicon-chip technology has made computing so economical that hundreds of millions of general-purpose computers are in use worldwide, helping people in business, industry, government and their personal lives. Given the current rate of technological development, this number could easily double over the next few years.

In beginning to study this text, you are starting on a challenging and rewarding educational path. As you proceed, if you would like to communicate with us, please send an e-mail to **`deitel@deitel.com`** or browse our World Wide Web sites at **`www.deitel.com`**, **`www.prenhall.com/deitel`** and **`www.InformIT.com/deitel`**. We hope that you enjoy learning C# through reading *C# How to Program*.

1.2 What Is a Computer?

A *computer* is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than those of human beings. For example, many of today's personal computers can perform hundreds of millions—even billions—of additions per second. A person operating a desk calculator might require decades to complete the same number of calculations that a powerful personal computer can perform in one second. (*Points to ponder*: How would you know whether the person had added the numbers correctly? How would you know whether the computer had added the numbers correctly?) Today's fastest *supercomputers* can perform hundreds of billions of additions per second—about as many calculations as hundreds of thousands of people could perform in one year! Trillion-instruction-per-second computers are already functioning in research laboratories!

Computers process *data* under the control of sets of instructions called *computer programs*. These programs guide computers through orderly sets of actions that are specified by individuals known as *computer programmers*.

A computer is composed of various devices (such as the keyboard, screen, mouse, disks, memory, CD-ROM and processing units) known as *hardware*. The programs that run on a computer are referred to as *software*. Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. Software-development costs, however, have been rising steadily, as programmers develop ever more powerful and complex applications without being able to improve significantly the technology of software development. In this book, you will learn proven software-development methods that can reduce software-development costs—top-down stepwise refinement, functionalization and object-oriented programming. Object-oriented programming is widely believed to be the significant breakthrough that can greatly enhance programmer productivity.

1.3 Computer Organization

Virtually every computer, regardless of differences in physical appearance, can be envisioned as being divided into six *logical units*, or sections:

1. *Input unit.* This “receiving” section of the computer obtains information (data and computer programs) from various *input devices*. The input unit then places this information at the disposal of the other units to facilitate the processing of the information. Today, most users enter information into computers via keyboards and mouse devices. Other input devices include microphones (for speaking to the computer), scanners (for scanning images) and digital cameras (for taking photographs and making videos).
2. *Output unit.* This “shipping” section of the computer takes information that the computer has processed and places it on various *output devices*, making the information available for use outside the computer. Computers can output information in various ways, including displaying the output on screens, playing it on audio/video devices, printing it on paper or using the output to control other devices.
3. *Memory unit.* This is the rapid-access, relatively low-capacity “warehouse” section of the computer, which facilitates the temporary storage of data. The memory unit retains information that has been entered through the input unit, enabling that information to be immediately available for processing. In addition, the unit retains processed information until that information can be transmitted to output devices. Often, the memory unit is called either *memory* or *primary memory*—*random access memory* (*RAM*) is an example of primary memory. Primary memory is usually volatile, which means that it is erased when the machine is powered off.
4. *Arithmetic and logic unit* (*ALU*). The ALU is the “manufacturing” section of the computer. It is responsible for the performance of calculations such as addition, subtraction, multiplication and division. It also contains decision mechanisms, allowing the computer to perform such tasks as determining whether two items stored in memory are equal.
5. *Central processing unit* (*CPU*). The CPU serves as the “administrative” section of the computer. This is the computer’s coordinator, responsible for supervising the operation of the other sections. The CPU alerts the input unit when information should be read into the memory unit, instructs the ALU about when to use information from the memory unit in calculations and tells the output unit when to send information from the memory unit to certain output devices.
6. *Secondary storage unit.* This unit is the long-term, high-capacity “warehousing” section of the computer. Secondary storage devices, such as hard drives and disks, normally hold programs or data that other units are not actively using; the computer then can retrieve this information when it is needed—hours, days, months or even years later. Information in secondary storage takes much longer to access than does information in primary memory. However, the price per unit of secondary storage is much less than the price per unit of primary memory. Secondary storage is usually *nonvolatile*—it retains information even when the computer is off.

1.4 Evolution of Operating Systems

Early computers were capable of performing only one *job* or *task* at a time. In this mode of computer operation, often called single-user *batch processing*, the computer runs one program at a time and processes data in groups called *batches*. Users of these early systems typically submitted their jobs to a computer center on decks of punched cards. Often, hours or even days elapsed before results were returned to the users' desks.

To make computer use more convenient, software systems called *operating systems* were developed. Early operating systems oversaw and managed computers' transitions between jobs. By minimizing the time it took for a computer operator to switch from one job to another, the operating system increased the total amount of work, or *throughput*, computers could process in a given time period.

As computers became more powerful, single-user batch processing became inefficient, because computers spent a great deal of time waiting for slow input/output devices to complete their tasks. Developers then looked to multiprogramming techniques, which enabled many tasks to *share* the resources of the computer to achieve better utilization. *Multiprogramming* involves the "simultaneous" operation of many jobs on a computer that splits its resources among those jobs. However, users of early multiprogramming operating systems still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several industry and university groups pioneered *timesharing* operating systems. Timesharing is a special type of multiprogramming that allows users to access a computer through *terminals* (devices with keyboards and screens). Dozens or even hundreds of people can use a timesharing computer system at once. It is important to note that the computer does not actually run all the users' requests simultaneously. Rather, it performs a small portion of one user's job and moves on to service the next user. However, because the computer does this so quickly, it can provide service to each user several times per second. This gives users' programs the appearance of running simultaneously. Timesharing offers major advantages over previous computing systems in that users receive prompt responses to requests, instead of waiting long periods to obtain results.

The UNIX operating system, which is now widely used for advanced computing, originated as an experimental timesharing operating system. Dennis Ritchie and Ken Thompson developed UNIX at Bell Laboratories beginning in the late 1960s and developed C as the language in which they wrote it. They freely distributed the source code to other programmers who wanted to use, modify and extend it. A large community of UNIX users quickly developed. The operating system grew as UNIX users contributed their own programs and tools. Through a collaborative effort among numerous researchers and developers, UNIX became a powerful and flexible operating system able to handle almost any type of task that a user required. Many versions of UNIX have evolved, including today's phenomenally popular *open-source* Linux operating system. Typically, the source code for open-source products is freely available over the Internet. This enables developers to learn from, validate and modify the source code. Often, open-source products require that developers publish any enhancements they make so the open-source community can continue to evolve those products.

1.5 Personal Computing, Distributed Computing and Client/Server Computing

In 1977, Apple Computer popularized the phenomenon of *personal computing*. Initially, it was a hobbyist's dream. However, the price of computers soon dropped so far that large numbers of people could buy them for personal or business use. In 1981, IBM, the world's largest computer vendor, introduced the IBM Personal Computer. Personal computing rapidly became legitimate in business, industry and government organizations.

The computers first pioneered by Apple and IBM were “stand-alone” units—people did their work on their own machines and transported disks back and forth to share information. (This process was often called “sneakernet.”) Although early personal computers were not powerful enough to timeshare several users, the machines could be linked together into computer networks, either over telephone lines or via *local area networks* (*LANs*) within an organization. These networks led to the *distributed computing* phenomenon, in which an organization’s computing is distributed over networks to the sites at which the work of the organization is performed, instead of being performed only at a central computer installation. Personal computers were powerful enough to handle both the computing requirements of individual users and the basic tasks involved in the electronic transfer of information between computers. *N-tier applications* split up an application over numerous distributed computers. For example, a *three-tier application* might have a user interface on one computer, business-logic processing on a second and a database on a third; all interact as the application runs.

Today’s most advanced personal computers are as powerful as the million-dollar machines of just two decades ago. High-powered desktop machines—called *workstations*—provide individual users with enormous capabilities. Information is easily shared across computer networks, in which computers called *servers* store programs and data that can be used by *client* computers distributed throughout the network. This type of configuration gave rise to the term *client/server computing*. Today’s popular operating systems, such as UNIX, Solaris, MacOS, Windows 2000, Windows XP and Linux, provide the kinds of capabilities discussed in this section.

1.6 Machine Languages, Assembly Languages and High-level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others that require intermediate *translation* steps. Although hundreds of computer languages are in use today, the diverse offerings can be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can understand only its own *machine language* directly. As the “natural language” of a particular computer, machine language is defined by the computer’s hardware design. Machine languages generally consist of streams of numbers (ultimately reduced to 1s and 0s) that instruct computers how to perform their most elementary operations. Machine languages are *machine-dependent*, which means that a particular machine language can be used on only one type of computer. The following section of a machine-language program, which adds *overtime* pay to *base pay* and stores the result in *gross pay*, demonstrates the incomprehensibility of machine language to the human reader.

```
+1300042774
+1400593419
+1200274027
```

As the popularity of computers increased, machine-language programming proved to be excessively slow, tedious and error prone. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the elementary operations of the computer. These abbreviations formed the basis of *assembly languages*. *Translator programs* called *assemblers* convert assembly language programs to machine language at computer speeds. The following section of an assembly-language program also adds *overtime pay* to *base pay* and stores the result in *gross pay*, but presents the steps more clearly to human readers than does its machine-language equivalent:

```
LOAD   BASEPAY
ADD    OVERPAY
STORE  GROSSPAY
```

Such code is clearer to humans but incomprehensible to computers until translated into machine language.

Although computer use increased rapidly with the advent of assembly languages, these languages still required many instructions to accomplish even the simplest tasks. To speed up the programming process, *high-level languages*, in which single statements accomplish substantial tasks, were developed. Translation programs called *compilers* convert high-level-language programs into machine language. High-level languages enable programmers to write instructions that look almost like everyday English and contain common mathematical notations. A payroll program written in a high-level language might contain a statement such as

```
grossPay = basePay + overTimePay
```

Obviously, programmers prefer high-level languages to either machine languages or assembly languages.

The compilation of a high-level language program into machine language can require a considerable amount of time. However, this problem was solved by the development of *interpreter* programs that can execute high-level language programs directly, bypassing the compilation step. Although programs that are already compiled execute faster than interpreted programs, interpreters are popular in program-development environments. In these environments, developers change programs frequently as they add new features and correct errors. Once a program is fully developed, a compiled version can be produced so that the program runs at maximum efficiency.

1.7 C, C++, Visual Basic .NET and Java™

As high-level languages develop, new offerings build on aspects of their predecessors. C++ evolved from C, which in turn evolved from two previous languages, BCPL and B. Martin Richards developed BCPL in 1967 as a language for writing operating systems, software and compilers. Ken Thompson modeled his language, B, after BCPL. In 1970, Thompson used B to create early versions of the UNIX operating system. Both BCPL and B were “typeless” languages, meaning that every data item occupied one “word” in memory. Using these languages, programmers assumed responsibility for treating each data item as a whole number or real number, for example.

The C language, which Dennis Ritchie evolved from B at Bell Laboratories, was originally implemented in 1973. Although C employs many of BCPL and B’s important concepts, it also offers data typing and other features. C first gained widespread recognition as a development language of the UNIX operating system. However, C is now available for most computers, and many of today's major operating systems are written in C or C++. C is a hardware-independent language, and, with careful design, it is possible to write C programs that are portable to most computers.

C++, an extension of C using elements from Simula 67, a simulation programming language, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that "spruce up" the C language, but, more importantly, it provides capabilities for *object-oriented programming (OOP).*

At a time when demand for new and more powerful software is soaring, the ability to build software quickly, correctly and economically remains an elusive goal. However, this problem can be addressed in part through the use of *objects,* or reusable software *components* that model items in the real world (see Section 1.11). Software developers are discovering that a modular, object-oriented approach to design and implementation can make software development groups much more productive than is possible via previous popular programming techniques, such as structured programming. Furthermore, object-oriented programs are often easier to understand, correct and modify.

In addition to C++, many other object-oriented languages have been developed. These include Smalltalk, which was created at Xerox's Palo Alto Research Center (PARC). Smalltalk is a pure object-oriented language, which means that literally everything is an object. C++ is a hybrid language—it is possible to program in a C-like style, an object-oriented style or both. Although some perceive this range of options as a benefit, most programmers today believe that it is best to program in a purely object-oriented manner.

Developing Microsoft Windows-based applications in languages such as C and C++, however, proved to be a difficult and cumbersome process. When Bill Gates founded Microsoft Corporation, he implemented *BASIC* on several early personal computers. BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a programming language developed in the mid-1960s by Professors John Kemeny and Thomas Kurtz of Dartmouth College as a language for writing simple programs. BASIC's primary purpose was to familiarize novices with programming techniques. The natural evolution from BASIC to Visual Basic was introduced in 1991 as a result of the development of the Microsoft Windows graphical user interface (GUI) in the late 1980s and the early 1990s.

Although Visual Basic is derived from the BASIC programming language, it is a distinctly different language that offers such powerful features as graphical user interfaces, event handling, access to the *Windows 32-bit Application Programming Interface* (*Win32 API*), object-oriented programming and error handling. Visual Basic is one of the most popular event-driven, visual programming interfaces.

The latest version of Visual Basic, called *Visual Basic .NET*[1], is designed for Microsoft's new programming platform, .NET. Earlier versions of Visual Basic provided object-oriented capabilities, but Visual Basic .NET offers enhanced object orientation and makes use of the powerful library of reusable software components in .NET.

Around the same time that Visual Basic was being developed, many individuals projected that intelligent consumer-electronic devices would be the next major market in which microprocessors would have a profound impact. Recognizing this, Sun Microsystems in 1991 funded an internal corporate research project code-named Green. The project resulted in the development of a language based on C and C++. Although the language's creator, James Gosling, called it Oak (after an oak tree outside his window at Sun), it was later discovered that a computer language called Oak already existed. When a group of Sun employees visited a local coffee place, the name Java was suggested, and it stuck.

---

[1] The reader interested in Visual Basic .NET may want to consider our book, *Visual Basic .NET How to Program, Second Edition.*

Unfortunately, the Green project ran into some difficulties. The marketplace for intelligent consumer-electronic devices was not developing as quickly as Sun had anticipated. Worse yet, a major contract for which Sun competed was awarded to another company. The project was, at this point, in danger of being canceled. By sheer good fortune, the World Wide Web exploded in popularity in 1993, and Sun saw immediate potential for using Java to design *dynamic content* (i.e., animated and interactive content) for Web pages.

Sun formally announced Java at a conference in May 1995. Ordinarily, an event like this would not generate much publicity. However, Java grabbed the immediate attention of the business community because of the new, widespread interest in the World Wide Web. Developers now use Java to create Web pages with dynamic content, to build large-scale enterprise applications, to enhance the functionality of World Wide Web servers (the computers that provide the content distributed to our Web browsers when we browse Web sites), to provide applications for consumer devices (e.g., cell phones, pagers and PDAs) and for many other purposes.

1.8 C#

The advancement of programming tools (e.g., C++ and Java) and consumer-electronic devices (e.g., cell phones) created problems and new requirements. The integration of software components from various languages proved difficult, and installation problems were common because new versions of shared components were incompatible with old software. Developers also discovered they needed Web-based applications that could be accessed and used via the Internet. As a result of mobile electronic device popularity, software developers realized that their clients were no longer restricted to desktop computers. Developers recognized the need for software that was accessible to anyone and available via almost any type of device. To address these needs, Microsoft announced its *.NET* (pronounced "dot-net") *initiative* and the *C#* (pronounced "C-Sharp") programming language.

The *.NET platform* is one over which Web-based applications can be distributed to a great variety of devices (even cell phones) and to desktop computers. The platform offers a new software-development model that allows applications created in disparate programming languages to communicate with each other. The C# programming language, developed at Microsoft by a team led by Anders Hejlsberg and Scott Wiltamuth, was designed specifically for the .NET platform as a language that would enable programmers to migrate easily to .NET. This migration is made easy due to the fact that C# has roots in C, C++ and Java, adapting the best features of each and adding new features of its own. Because C# has been built upon such widely used and well-developed languages, programmers will find learning C# to be easy and enjoyable.

C# is an event-driven, fully object-oriented, visual programming language in which programs are created using an *Integrated Development Environment* (*IDE*). With the IDE, a programmer can create, run, test and debug C# programs conveniently, thereby reducing the time it takes to produce a working program to a fraction of the time it would have taken without using the IDE. The process of rapidly creating an application using an IDE is typically referred to as *Rapid Application Development (RAD)*.

C# also enables a new degree of language interoperability: Software components from different languages can interact as never before. Developers can package even old software to work with new C# programs. In addition, C# applications can interact via the Internet, using industry standards such as the Simple Object Access Protocol (SOAP) and XML, which we discuss in Chapter 18, Extensible Markup Language (XML). The programming advances embodied in .NET and C# will lead to a new style of programming, in which applications are created from building blocks available over the Internet.

1.9 Other High-level Languages

Although hundreds of high-level languages have been developed, only a few have achieved broad acceptance. This section overviews several languages that, like BASIC, are long-standing and popular high-level languages. IBM Corporation developed Fortran (FORmula TRANslator) between 1954 and 1957 to create scientific and engineering applications that require complex mathematical computations. Fortran is still widely used.

COBOL (COmmon Business Oriented Language) was developed in 1959 by a group of computer manufacturers in conjunction with government and industrial computer users. COBOL is used primarily for commercial applications that require the precise and efficient manipulation of large amounts of data. A considerable portion of today’s business software is still programmed in COBOL. Approximately one million programmers are actively writing in COBOL.

Pascal was designed in the late 1960s by Professor Nicklaus Wirth and was intended for academic use. We explore Pascal in the next section.

1.10 Structured Programming

During the 1960s, many large software-development efforts encountered severe difficulties. Development typically ran behind schedule, costs greatly exceeded budgets and the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity, intended to address these issues, resulted in the evolution of *structured programming*—a disciplined approach to the creation of programs that are clear, demonstrably correct and easy to modify.

One of the more tangible results of this research was the development of the *Pascal* programming language in 1971. Pascal, named after the seventeenth-century mathematician and philosopher Blaise Pascal, was designed for teaching structured programming in academic environments and rapidly became the preferred introductory programming language in most universities. Unfortunately, because the language lacked many features needed to make it useful in commercial, industrial and government applications, it was not widely accepted in these environments. By contrast, C, which also arose from research on structured programming, did not have the limitations of Pascal, and programmers quickly adopted it.

The *Ada* programming language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of programming languages were being used to produce DOD’s massive command-and-control software systems. DOD wanted a single language that would meet its needs. Pascal was chosen as a base, but the final Ada language is quite different from Pascal. The language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is generally credited with writing the world's first computer program, in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada is *multitasking*, which allows programmers to specify that many activities are to occur in parallel. As we will see in Chapter 14, C# offers a similar capability, called *multithreading*.

1.11 Key Software Trend: Object Technology

One of the authors, HMD, remembers the great frustration felt in the 1960s by software-development organizations, especially those developing large-scale projects. During the summers of his undergraduate years, HMD had the privilege of working at a leading computer vendor on the teams developing time-sharing, virtual-memory operating systems. It was a great experience for a college student, but, in the summer of 1967, reality set in. The company "decommitted" from producing as a commercial product the particular system that hundreds of people had been working on for several years. It was difficult to get this software right. Software is "complex stuff."

As the benefits of structured programming (and the related disciplines of *structured systems analysis and design*) were realized in the 1970s, improved software technology did begin to appear. However, it was not until the technology of object-oriented programming became widely used in the 1980s and 1990s that software developers finally felt they had the necessary tools to improve the software-development process dramatically.

Actually, object technology dates back to at least the mid-1960s, but no broad-based programming language incorporated the technology until C++. Although not strictly an object-oriented language, C++ absorbed the capabilities of C and incorporated Simula's ability to create and manipulate objects. C++ was never intended for widespread use beyond the research laboratories at AT&T, but grass-roots support rapidly developed for the hybrid language.

What are objects, and why are they special? Object technology is a packaging scheme that facilitates the creation of meaningful software units. These units are large and focused on particular applications areas. There are date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects and so on. In fact, almost any noun can be reasonably represented as a software object. Objects have *properties* (i.e., *attributes*, such as color, size and weight) and perform *actions* (i.e., *behaviors*, such as moving, sleeping or drawing). Classes represent groups of related objects. For example, all cars belong to the "car" class, even though individual cars vary in make, model, color and options packages. A class specifies the general format of its objects; the properties and actions available to an object depend on its class.

We live in a world of objects. Just look around you—there are cars, planes, people, animals, buildings, traffic lights, elevators and so on. Before object-oriented languages appeared, *procedural programming languages* (such as Fortran, Pascal, BASIC and C) focused on actions (verbs) rather than things or objects (nouns). We live in a world of objects, but earlier programming languages forced individuals to program primarily with verbs. This paradigm shift made program writing a bit awkward. However, with the advent of popular object-oriented languages, such as C++, Java and C#, programmers can program in an object-oriented manner that reflects the way in which they perceive the world. This process, which seems more natural than procedural programming, has resulted in significant productivity gains.

One of the key problems with procedural programming is that the program units created do not mirror real-world entities effectively and therefore are not particularly reusable. Programmers often write and rewrite similar software for various projects. This wastes precious time and money as people repeatedly “reinvent the wheel.” With object technology, properly designed software entities (called objects) can be reused on future projects. Using libraries of reusable componentry can greatly reduce the amount of effort required to implement certain kinds of systems (as compared to the effort that would be required to reinvent these capabilities in new projects). C# programmers use the .NET Framework Class Library (known commonly as the FCL).

Some organizations report that software reusability is not, in fact, the key benefit of object-oriented programming. Rather, they indicate that object-oriented programming tends to produce software that is more understandable because it is better organized and has fewer maintenance requirements. As much as 80 percent of software costs are not associated with the original efforts to develop the software, but instead are related to the continued evolution and maintenance of that software throughout its lifetime. Object orientation allows programmers to abstract the details of software and focus on the “big picture.” Rather than worrying about minute details, the programmer can focus on the behaviors and interactions of objects. A roadmap that showed every tree, house and driveway would be difficult, if not impossible, to read. When such details are removed and only the essential information (roads) remains, the map becomes easier to understand. In the same way, a program that is divided into objects is easy to understand, modify and update because it hides much of the detail. It is clear that object-oriented programming will be the key programming methodology for at least the next decade.

Software Engineering Observation 1.1

Use a building-block approach to creating programs. By using existing pieces in new projects, programmers avoid reinventing the wheel. This is called software reuse, *and it is central to object-oriented programming.*

[*Note*: We will include many of these *Software Engineering Observations* throughout the text to explain concepts that affect and improve the overall architecture and quality of a software system and, particularly, of large software systems. We will also highlight *Good Programming Practices* (practices that can help you write programs that are clearer, more understandable, more maintainable and easier to test and debug), *Common Programming Errors* (problems to watch for to ensure that you do not make these same errors in your programs), *Performance Tips* (techniques that will help you write programs that run faster and use less memory), *Portability Tips* (techniques that will help you write programs that can run, with little or no modification, on a variety of computers), *Testing and Debugging Tips* (techniques that will help you remove bugs from your programs and, more importantly, write bug-free programs in the first place) and *Look-and-Feel Observations* (techniques that will help you design the “look and feel” of your graphical user interfaces for appearance and ease of use). Many of these techniques and practices are only guidelines; you will, no doubt, develop your own preferred programming style.]

The advantage of creating your own code is that you will know exactly how it works. The code will be yours to examine, modify and improve. The disadvantage is the time and effort that goes into designing, developing and testing new code.

Performance Tip 1.1

Reusing proven code components instead of writing your own versions can improve program performance, because these components normally are written to perform efficiently.

Software Engineering Observation 1.2

Extensive class libraries of reusable software components are available over the Internet and the World Wide Web; many are offered free of charge.

1.12 Hardware Trends

Every year, people generally expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the costs of hardware supporting these technologies. For many decades, and continuing into the foreseeable future, hardware costs have fallen rapidly, if not precipitously. Every year or two, the capacities of computers approximately double.[2] This is especially true in relation to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) computers have to hold programs and data over longer periods of time and their processor speeds—the speeds at which computers execute their programs (i.e., do their work). Similar improvements have occurred in the communications field, in which costs have plummeted as enormous demand for communications bandwidth (i.e., information-carrying capacity) has attracted tremendous competition. We know of no other fields in which technology moves so quickly and costs fall so rapidly. Such phenomenal improvement in the computing and communications fields is truly fostering the so-called *Information Revolution*.

When computer use exploded in the 1960s and 1970s, many discussed the dramatic improvements in human productivity that computing and communications would cause. However, these improvements did not materialize. Organizations were spending vast sums of capital on computers and employing them effectively, but without fully realizing the expected productivity gains. The invention of microprocessor chip technology and its wide deployment in the late 1970s and 1980s laid the groundwork for the productivity improvements that individuals and businesses have achieved in recent years.

1.13 History of the Internet and World Wide Web

In the late 1960s, one of the authors (HMD) was a graduate student at MIT. His research at MIT's Project Mac (now the Laboratory for Computer Science—the home of the World Wide Web Consortium) was funded by ARPA—the Advanced Research Projects Agency of the Department of Defense. ARPA sponsored a conference at which several dozen ARPA-funded graduate students were brought together at the University of Illinois at Urbana-Champaign to meet and share ideas. During this conference, ARPA rolled out the blueprints for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at a then-stunning 56 Kbps (1 Kbps is equal to 1,024 bits per second), at a time when most people (of the few who had access to networking technologies) were connecting over telephone lines to computers at a rate of 110 bits per second.

---

[2] This often is called *Moore's Law*.

HMD vividly recalls the excitement at that conference. Researchers at Harvard talked about communicating with the Univac 1108 "supercomputer," which was located across the country at the University of Utah, to handle calculations related to their computer graphics research. Many other intriguing possibilities were discussed. Academic research was about to take a giant leap forward. Shortly after this conference, ARPA proceeded to implement what quickly became called the *ARPAnet,* the grandparent of today's *Internet.*

Things worked out differently from the original plan. Although the ARPAnet did enable researchers to network their computers, its chief benefit proved to be the capability for quick and easy communication via what came to be known as *electronic mail* (*e-mail*). This is true even on today's Internet, with e-mail, instant messaging and file transfer facilitating communications among hundreds of millions of people worldwide.

The network was designed to operate without centralized control. This meant that, if a portion of the network should fail, the remaining working portions would still be able to route data packets from senders to receivers over alternative paths.

The protocol (i.e., set of rules) for communicating over the ARPAnet became known as the *Transmission Control Protocol (TCP)*. TCP ensured that messages were properly routed from sender to receiver and that those messages arrived intact.

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks to facilitate both intra-organization (i.e., within the organization) and inter-organization (i.e., between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these diverse products to communicate with each other. ARPA accomplished this by developing the *Internet Protocol* (*IP),* which created a true "network of networks," the current architecture of the Internet. The combined set of protocols is now commonly called *TCP/IP*.

Initially, use of the Internet was limited to universities and research institutions; later, the military adopted the technology. Eventually, the government decided to allow access to the Internet for commercial purposes. When this decision was made, there was resentment among the research and military communities—it was felt that response times would become poor as "the Net" became saturated with so many users.

In fact, the opposite has occurred. Businesses rapidly realized that, by making effective use of the Internet, they could refine their operations and offer new and better services to their clients. Companies started spending vast amounts of money to develop and enhance their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. The result is that *bandwidth* (i.e., the information-carrying capacity of communications lines) on the Internet has increased tremendously, while hardware costs have plummeted. It is widely believed that the Internet played a significant role in the economic growth that many industrialized nations experienced over the last decade.

The *World Wide Web* allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animations, audios and/or videos) on almost any subject. Even though the Internet was developed more than three decades ago, the introduction of the World Wide Web (WWW) was a relatively recent event. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began to develop a technology for sharing information via hyperlinked text documents. Basing the new language on the well-established *Standard Generalized Markup Language* (*SGML*)—a standard for business data interchange—Berners-Lee called his invention the *HyperText*

Markup Language (*HTML*). He also wrote communication protocols to form the backbone of his new hypertext information system, which he referred to as the *World Wide Web*.

Historians will surely list the Internet and the World Wide Web among the most important and profound creations of humankind. In the past, most computer applications ran on "stand-alone" computers (computers that were not connected to one another). Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet and World Wide Web merge computing and communications technologies, expediting and simplifying our work. They make information instantly and conveniently accessible to large numbers of people. They enable individuals and small businesses to achieve worldwide exposure. They are profoundly changing the way we do business and conduct our personal lives.

1.14 World Wide Web Consortium (W3C)

In October 1994, Tim Berners-Lee founded an organization, called the *World Wide Web Consortium* (*W3C*), that is devoted to developing nonproprietary, interoperable technologies for the World Wide Web. One of the W3C's primary goals is to make the Web universally accessible—regardless of disabilities, language or culture.

The W3C is also a standardization organization and is comprised of three *hosts*—the Massachusetts Institute of Technology (MIT), France's INRIA (Institut National de Recherche en Informatique et Automatique) and Keio University of Japan—and over 400 members, including Deitel & Associates, Inc. Members provide the primary financing for the W3C and help provide the strategic direction of the Consortium. To learn more about the W3C, visit **`www.w3.org`**.

Web technologies standardized by the W3C are called *Recommendations*. Current W3C Recommendations include *Extensible HyperText Markup Language (XHTML™)*, *Cascading Style Sheets (CSS™)* and the *Extensible Markup Language (XML)*. Recommendations are not actual software products, but documents that specify the role, syntax and rules of a technology. Before becoming a W3C Recommendation, a document passes through three major phases: *Working Draft*—which, as its name implies, specifies an evolving draft; *Candidate Recommendation*—a stable version of the document that industry can begin to implement; and *Proposed Recommendation*—a Candidate Recommendation that is considered mature (i.e., has been implemented and tested over a period of time) and is ready to be considered for W3C Recommendation status. For detailed information about the W3C Recommendation track, see "6.2 The W3C Recommendation track" at

```
www.w3.org/Consortium/Process/Process-19991111/
process.html#RecsCR
```

1.15 Extensible Markup Language (XML)

As the popularity of the Web exploded, HTML's limitations became apparent. HTML's lack of *extensibility* (the ability to change or add features) frustrated developers, and its ambiguous definition allowed erroneous HTML to proliferate. In response to these problems, the W3C added limited extensibility to HTML. This was, however, only a temporary solution—the need for a standardized, fully extensible and structurally strict language was apparent. As a result, XML was developed by the W3C. XML combines the power and extensibility of its parent language, Standard Generalized Markup Language (SGML), with the simplicity that the Web community demands. At the same time, the W3C began developing XML-based standards for style sheets and advanced hyperlinking. *Extensible Stylesheet Language* (*XSL*) incorporates elements of both Cascading Style Sheets (CSS), which is used to format HTML documents and *Document Style and Semantics Specification Language* (*DSSSL*), which is used to format SGML documents. Similarly, the *Extensible Linking Language* (*XLink*) combines ideas from *HyTime* and the *Text Encoding Initiative* (*TEI*), to provide extensible linking of resources.

Data independence, the separation of content from its presentation, is the essential characteristic of XML. Because an XML document describes data, any application conceivably can process an XML document. Recognizing this, software developers are integrating XML into their applications to improve Web functionality and interoperability. XML's flexibility and power make it perfect for the middle tier of client/server systems, which must interact with a wide variety of clients. Much of the processing that was once limited to server computers now can be performed by client computers, because XML's semantic and structural information enables it to be manipulated by any application that can process text. This reduces server loads and network traffic, resulting in a faster, more efficient Web.

XML is not limited to Web applications. Increasingly, XML is being employed in databases—the structure of an XML document enables it to be integrated easily with database applications. As applications become more Web enabled, it seems likely that XML will become the universal technology for data representation. All applications employing XML would be able to communicate, provided that they could understand each other's XML markup, or *vocabulary*.

Simple Object Access Protocol (SOAP) is a technology for the distribution of objects (marked up as XML) over the Internet. Developed primarily by Microsoft and DevelopMentor, SOAP provides a framework for expressing application semantics, encoding that data and packaging it in modules. SOAP has three parts: The *envelope*, which describes the content and intended recipient of a SOAP message; the SOAP *encoding rules*, which are XML-based; and the SOAP *Remote Procedure Call* (*RPC*) *representation* for commanding other computers to perform a task. Microsoft .NET (discussed in the next two sections) uses XML and SOAP to mark up and transfer data over the Internet. XML and SOAP are at the core of .NET—they allow software components to interoperate (i.e., communicate easily with one another). SOAP is supported by many platforms, because of its foundations in XML and HTTP. We discuss XML in Chapter 18, Extensible Markup Language (XML) and SOAP in Chapter 21, ASP .NET and Web Services.

1.16 Introduction to Microsoft .NET

In June 2000, Microsoft announced its *.NET initiative*, a broad new vision for embracing the Internet and the Web in the development, engineering and use of software. One key aspect of the .NET strategy is its independence from a specific language or platform. Rather than forcing developers to use a single programming language, developers can create a .NET application in any .NET-compatible language. Programmers can contribute to the same software project, writing code in the .NET languages (such as C#, Visual C++ .NET, Visual Basic .NET and many others) in which they are most competent. Part of the initiative includes Microsoft's *Active Server Pages (ASP) .NET* technology, which allows programmers to create applications for the Web.

The .NET architecture can exist on multiple platforms, further extending the portability of .NET programs. In addition, the .NET strategy involves a new program-development process that could change the way programs are written and executed, leading to increased productivity.

A key component of the .NET architecture is *Web services*, which are applications that can be used over the Internet. Clients and other applications can use these Web services as reusable building blocks. One example of a Web service is Dollar Rent a Car's reservation system.[3] An airline partner wanted to enable customers to make rental-car reservations from the airline's Web site. To do so, the airline needed to access Dollar's reservation system. In response, Dollar created a Web service that allowed the airline to access Dollar's database and make reservations. Web services enable the two companies to communicate over the Web, even though the airline uses UNIX systems and Dollar uses Microsoft Windows. Dollar could have created a one-time solution for that particular airline, but the company would not have been able to reuse such a customized system. By creating a Web service, Dollar can allow other airlines or hotels to use its reservation system without creating a custom program for each relationship.

The .NET strategy extends the concept of software reuse to the Internet, allowing programmers to concentrate on their specialties without having to implement every component of every application. Instead, companies can buy Web services and devote their time and energy to developing their products. The .NET strategy further extends the concept of software reuse to the Internet by allowing programmers to concentrate on their specialties without having to implement every component. Visual programming (discussed in Chapter 2) has become popular, because it enables programmers to create applications easily, using such prepackaged components as buttons, text boxes and scrollbars. Similarly, programmers may create an application using Web services for databases, security, authentication, data storage and language translation without having to know the internal details of those components.

The .NET strategy incorporates the idea of software reuse. When companies link their products in this way, a new user experience emerges. For example, a single application could manage bill payments, tax refunds, loans and investments, using Web services from various companies. An online merchant could buy Web services for online credit-card payments, user authentication, network security and inventory databases to create an e-commerce Web site.

The keys to this interaction are XML and SOAP, which enable Web services to communicate. XML gives meaning to data, and SOAP is the protocol that allows Web services to communicate easily with one another. XML and SOAP act as the "glue" that combines various Web services to form applications.

Universal data access is another essential concept in the .NET strategy. If two copies of a file exist (such as on a personal and a company computer), the less recent version must constantly be updated—this is called file *synchronization*. If the separate versions of the file are different, they are *unsynchronized*, a situation that could lead to errors. Under .NET, data could reside in one central location rather than on separate systems. Any Internet-connected device could access the data (under tight control, of course), which would then be formatted appropriately for use or display on the accessing device. Thus, the same document could be seen and edited on a desktop PC, a PDA, a cell phone or other device. Users would not need to synchronize the information, because it would be fully up-to-date in a central area.

---

[3] Microsoft Corporation, "Dollar Rent A Car E-Commerce Case Study on Microsoft Business," 1 July 2001 **`<www.microsoft.com/BUSINESS/casestudies/b2c/dollarrentacar.asp>`**.

Microsoft's *HailStorm Web services* facilitate such data organization.[4] HailStorm allows users to store data so that it is accessible from any HailStorm-compatible device (such as a PDA, desktop computer or cell phone). HailStorm offers a suite of services, such as an address book, e-mail, document storage, calendars and a digital wallet. Third-party Web services also can interact with HailStorm—users can be notified when they win online auctions or have their calendars updated if their planes arrive late. Information can be accessed from anywhere and cannot become unsynchronized. Privacy concerns increase, though, because all of a user's data resides in one location. Microsoft has addressed this issue by giving users control over their data. Users must authorize access to their data and specify the duration of that access.

Microsoft plans to create Internet-based client applications. For example, software could be distributed over the Internet on a *subscription basis*, enabling immediate corrections, updates and communication with other applications over the Internet. HailStorm provides basic services at no charge and users can pay via subscription for more advanced features.

The .NET strategy is an immense undertaking. We discuss various aspects of .NET throughout this book. Additional information is available on Microsoft's Web site (**`www.microsoft.com/net`**).

1.17 .NET Framework and the Common Language Runtime

The Microsoft® *.NET Framework* is at the heart of the .NET strategy. This framework manages and executes applications and Web services, contains a class library (called the *Framework Class Library* or *FCL*), enforces security and provides many other programming capabilities. The details of the .NET Framework are found in the *Common Language Specification* (*CLS*), which contains information about the storage of data types, objects and so on. The CLS has been submitted for standardization to ECMA (the European Computer Manufacturers Association), making it easier to create the .NET Framework for other platforms. This is like publishing the blueprints of the framework—anyone can build it, following the specifications. Currently, the .NET Framework exists only for the Windows platform, although a version is under development for the FreeBSD operating system.[5] The FreeBSD project provides a freely available and open-source UNIX-like operating system that is based on that UC Berkeley's *Berkeley System Distribution* (BSD).

The *Common Language Runtime (CLR)* is another central part of the .NET Framework—it executes C# programs. Programs are compiled into machine-specific instructions in two steps. First, the program is compiled into *Microsoft Intermediate Language (MSIL)*, which defines instructions for the CLR. Code converted into MSIL from other languages and sources can be woven together by the CLR. Then, another compiler in the CLR compiles the MSIL into machine code (for a particular platform), creating a single application.

---

[4] Microsoft Corporation, "Building User-Centric Experiences: An Introduction to Microsoft HailStorm," 30 July 2001 <**`http://www.microsoft.com/net/hailstorm.asp`**>.

[5] Microsoft Corporation, "The Microsoft Shared Source C# and CLI Specifications," 30 July 2001 **`<http://www.microsoft.com/net/sharedsourcewp.asp>`**.

Why bother having the extra step of converting from C# to MSIL, instead of compiling directly into machine language? The key reasons are portability between operating systems, interoperability between languages and execution-management features such as memory management and security.

If the .NET Framework exists (and is installed) for a platform, that platform can run any .NET program. The ability of a program to run (without modification) across multiple platforms is known as *platform independence*. Code written once can be used on another machine without modification, saving both time and money. In addition, software can target a wider audience—previously, companies had to decide whether converting their programs to different platforms (sometimes called *porting*) was worth the cost. With .NET, porting is no longer an issue.

The .NET Framework also provides a high level of *language interoperability*. Programs written in different languages are all compiled into MSIL—the different parts can be combined to create a single, unified program. MSIL allows the .NET Framework to be *language independent*, because .NET programs are not tied to a particular programming language. Any language that can be compiled into MSIL is called a *.NET-compliant language*. Figure 1.1 lists many of the current languages that support the .NET platform.[6]

Language interoperability offers many benefits to software companies. C#, Visual Basic .NET and Visual C++ .NET developers, for example, can work side-by-side on the same project without having to learn another programming language—all their code compiles into MSIL and links together to form one program. In addition, the .NET Framework can package old and new components to work together. This allows companies to reuse the code that they have spent years developing and integrate it with the new .NET code that they write. Integration is crucial, because companies cannot migrate easily to .NET unless they can stay productive, using their existing developers and software.

Programming Languages
APL
C#
COBOL
Component Pascal
Curriculum
Eiffel
Fortran
Haskell
Java
JScript
Mercury
Oberon
Oz
Pascal
Perl
Python
RPG
Scheme
Smalltalk
Standard ML
Visual Basic .NET
Visual C++ .NET

Fig. 1.1 .NET Languages.

---

[6] Table information from Microsoft Web site, `www.microsoft.com`.

Another benefit of the .NET Framework is the CLR's execution-management features. The CLR manages memory, security and other features, relieving the programmer of these responsibilities. With languages like C++, programmers must take memory management into their own hands. This leads to problems if programmers request memory and never return it—programs could consume all available memory, which would prevent applications from running. By managing the program's memory, the .NET Framework allows programmers to concentrate on program logic.

The .NET Framework also provides programmers with a huge library of classes. This library, called the Framework Class Library (FCL), can be used by any .NET language. The FCL contains a variety of reusable components, saving programmers the trouble of creating new components. This book explains how to develop .NET software with C#. Steve Ballmer, Microsoft's CEO, stated in May 2001 that Microsoft was "betting the company" on .NET. Such a dramatic commitment surely indicates a bright future for C# and its community of developers.

1.18 Tour of the Book

In this section, we tour the chapters and appendices of *C# How to Program*. In addition to the topics presented in each chapter, several of the chapters contain an Internet and World Wide Web Resources section that lists additional sources from which readers can enhance their knowledge of C# programming.

Chapter 1—Introduction to Computers, Internet, World Wide Web and C#

The first chapter familiarizes the reader with what computers are, how they work and how they are programmed. We explain the evolution of programming languages, from their origins in machine languages to the development of high-level, object-oriented languages. We overview the history of the Internet, World Wide Web and various technologies (such as HTTP, SOAP and XML) that have led to advances in how computers are used. We then discuss the development of the C# programming language and the Microsoft .NET initiative, including Web services. We explore the impact of .NET on software development and conclude by touring the remainder of the book.

Chapter 2—Introduction to the Visual Studio® .NET IDE

Chapter 2 introduces Microsoft Visual Studio .NET, an *integrated development environment* (*IDE*) for the creation of C# programs. Visual Studio .NET enables *visual programming*, in which *controls* (such as buttons or text boxes) are "dragged" and "dropped" into place, rather than added by typing code. Visual programming has led to greatly increased productivity of software developers because it eliminates many of the tedious tasks that programmers face. For example, object properties (information such as height and color) can be modified through Visual Studio .NET windows, allowing changes to be made quickly and causing the results to appear immediately on the screen. Rather than having to guess how the GUI will appear while writing a program, programmers view the GUI exactly as it will appear when the finished program runs. Visual Studio .NET also contains advanced tools for debugging, documenting and writing code. The chapter presents features of Visual Studio .NET, including its key windows, toolbox and help features and overviews the process of compiling and running programs. We provide an example of the capabilities of Visual Studio .NET by using it to create a simple Windows application without typing a single line of code.

Chapter 3—Introduction to C# Programming

This chapter introduces readers to our LIVE-CODE™ approach. Every concept is presented in the context of a complete working C# program and is followed by one or more sample outputs depicting the program's execution. In our first example, we print a line of text and carefully discuss each line of code. We then discuss fundamental tasks, such as how a program inputs data from its users and how to write arithmetic expressions. The chapter's last example demonstrates how to print a variety of character strings in a window called a message box.

Chapter 4—Control Structures: Part 1

This chapter formally introduces the principles of structured programming, a set of techniques that will help the reader develop clear, understandable, maintainable programs throughout the text. The first part of this chapter presents program-development and problem-solving techniques. The chapter demonstrates how to transform a written specification to a program by using such techniques as *pseudocode* and *top-down, stepwise refinement*. We then progress through the entire process, from developing a problem statement into a working C# program. The notion of algorithms is also discussed. We build on information presented in the previous chapter to create programs that are interactive (i.e., they change their behavior to suit user-supplied inputs). The chapter then introduces the use of control structures that affect the sequence in which statements are executed. Control structures produce programs that are easily understood, debugged and maintained. We discuss the three forms of program control—sequence, selection and repetition—focusing on the **`if`**/**`then`** and **`while`** control structures. Flowcharts (i.e., graphical representations of algorithms) appear throughout the chapter, reinforcing and augmenting the explanations.

Chapter 5—Control Structures: Part 2

Chapter 5 introduces more complex control structures and the logical operators. It uses flowcharts to illustrate the flow of control through each control structure, including the **`for`**, **`do`**/**`while`** and **`switch`** structures. We explain the **`break`** and **`continue`** statements and the logical operators. Examples include calculating compound interest and printing the distribution of grades on an exam (with some simple error checking). The chapter concludes with a structured programming summary, including each of C#'s control structures. The techniques discussed in Chapters 4 and 5 constitute a large part of what has been taught traditionally under the topic of structured programming.

Chapter 6—Methods

A *method* allows the programmer to create a block of code that can be called upon from various points in a program. Groups of related methods can be separated into functional blocks (classes), using the "divide and conquer" strategy. Programs are divided into simple components that interact in straightforward ways. We discuss how to create our own methods that can take input, perform calculations and return output. We examine the .NET library's **`Math`** class, which contains methods (i.e., methods in a class) for performing complex calculations (e.g., trigonometric and logarithmic calculations). *Recursive* methods (methods that call themselves) and method overloading, which allows multiple methods to have the same name, are introduced. We demonstrate overloading by creating two **`Square`** methods that take an integer (i.e., whole number) and a floating-point number (i.e., a number with a decimal point), respectively. To conclude the chapter, we create a graphical simulation of the "craps" dice game, using the random-number generation techniques presented in the chapter.

Chapter 7—Arrays

Chapter 7 discusses arrays, our first data structures. (Chapter 24 discusses the topic of data structures in depth.) Data structures are crucial to storing, sorting, searching and manipulating large amounts of information. *Arrays* are groups of related data items that allow the programmer to access any element directly. Rather than creating 100 separate variables that are all related in some way, the programmer instead can create an array of 100 elements and access these elements by their location in the array. We discuss how to declare and allocate arrays, and we build on the techniques of the previous chapter by passing arrays to methods. In addition, we discuss how to pass a variable number of arguments to methods. Chapters 4 and 5 provide essential background for the discussion of arrays, because repetition structures are used to iterate through elements in the array. The combination of these concepts helps the reader create highly-structured and well-organized programs. We then demonstrate how to sort and search arrays. We discuss multidimensional arrays (both rectangular and jagged), which can be used to store tables of data.

Chapter 8—Object-Based Programming

Chapter 8 serves as our introduction into the powerful concepts of objects and *classes* (classes are programmer-defined types). As mentioned in Chapter 1, object technology has led to considerable improvements in software development, allowing programmers to create reusable components. In addition, objects allow programs to be organized in natural and intuitive ways. In this chapter, we present the fundamentals of object-based programming, such as encapsulation, data abstraction and abstract data types (ADTs). These techniques hide the details of components so that the programmer can concentrate on the “big picture.” To demonstrate these concepts, we create a time class, which displays the time in standard and military formats. Other topics examined include abstraction, composition, reusability and inheritance. We overview how to create reusable software components with assemblies, namespaces and Dynamic Link Library (DLL) files. You will learn how to create classes like those in the Framework Class Library. Other C# features discussed include properties and the **`readonly`** and **`const`** keywords. This chapter lays the groundwork for the next two chapters, which introduce object-oriented programming.

Chapter 9—Object-Oriented Programming: Inheritance

In this chapter, we discuss inheritance—a form of software reusability in which classes (called *derived classes*) are created by absorbing attributes and methods of existing classes (called *base classes*). The inherited class (i.e., the derived class) can contain additional attributes and methods. We show how finding the commonality between classes of objects can reduce the amount of work it takes to build large software systems. These proven techniques help programmers create and maintain software systems. A detailed case study demonstrates software reuse and good programming techniques by finding the commonality among a three-level inheritance hierarchy: the point, circle and cylinder classes. We discuss the software engineering benefits of object-oriented programming. The reader learns important object-oriented programming fundamentals, such as creating and extending customized classes.

Chapter 10—Object-Oriented Programming: Polymorphism

Chapter 10 continues our formal introduction of object-oriented programming. We discuss polymorphic programming and its advantages. *Polymorphism* permits classes to be treated in a general manner, allowing the same method call to act differently depending on context (e.g., “move” messages sent to a bird and a fish result in dramatically different types of action—a bird flies and a fish swims). In addition to treating existing classes in a general manner, polymorphism allows new classes to be added to a system easily. We identify situations in which polymorphism is useful. A payroll system case study demonstrates polymorphism—the system determines the wages for each employee differently to suit the type of employee (bosses paid fixed salaries, hourly workers paid by the hour, commission workers who receive a base salary plus commission and piece workers who are paid per item produced). These programming techniques and those of the previous chapter allow the programmer to create extensible and reusable software components.

Chapter 11—Exception Handling

Exception handling is one of the most important topics in C# from the standpoint of building mission-critical and business-critical applications. People can enter incorrect data, data can be corrupted and clients can try to access records that do not exist or are restricted. A simple division-by-zero error may cause a calculator program to crash, but what if such an error occurs in the navigation system of a flying airplane? Programmers must deal with these situations, because in some cases, the results of program failure could be disastrous. Programmers need to know how to recognize the errors (*exceptions*) that could occur in software components and handle those exceptions effectively, allowing programs to deal with problems and continue executing instead of “crashing.” This chapter overviews the proper use of exception handling and various exception-handling techniques. We cover the details of C# exception handling, the termination model of exception handling, throwing and catching exceptions, and library class **`Exception`**. Programmers who construct software systems from reusable components built by other programmers often deal with the exceptions that those components may throw.

Chapter 12—Graphical User Interface Concepts: Part 1

Chapter 12 explains how to add graphical user interfaces (GUIs) to our programs, providing a professional look and feel. By using the techniques of rapid application development (RAD), we can create a GUI from reusable components, rather than explicitly programming every detail. The Visual Studio .NET IDE makes developing GUIs even easier by allowing the programmer to position components in a window through so-called visual programming. We discuss how to construct user interfaces with *Windows Forms* GUI components such as labels, buttons, text boxes, scroll bars and picture boxes. We also introduce *events*, which are messages sent by a program to signal to an object or a set of objects that an action has occurred. Events are most commonly used to signal user interactions with GUI components, but also can signal internal actions in a program. We overview event handling and discuss how to handle events specific to controls, the keyboard and the mouse. Tips are included throughout the chapter to help the programmer create visually appealing, well-organized and consistent GUIs.

Chapter 13—Graphical User Interface Concepts: Part 2

Chapter 13 introduces more complex GUI components, including menus, link labels, panels, list boxes, combo boxes and tab controls. In a challenging exercise, readers create an application that displays a disk drive’s directory structure in a tree—similar to that created by Windows Explorer. The *Multiple Document Interface* (*MDI*) is presented, which allows multiple documents (i.e., forms) to be open simultaneously in a single GUI. We conclude with a discussion of how to create custom controls by combining existing controls. The techniques presented in this chapter allow readers to create sophisticated and well-organized GUIs, adding style and usability to their applications.

Chapter 14—Multithreading

We have come to expect much from our applications. We want to download files from the Internet, listen to music, print documents and browse the Web—all at the same time! To do this, we need a technique called *multithreading*, which allows applications to perform multiple activities concurrently. C# includes built-in capabilities to enable multithreaded applications, while shielding programmers from complex details. C# is better equipped to deal with more sophisticated multimedia, network-based and multiprocessor-based applications than other languages that do not have multithreading features. This chapter overviews the built-in threading classes of C# and covers threads, thread life-cycles, time-slicing, scheduling and priorities. We analyze the producer-consumer relationship, thread synchronization and circular buffers. This chapter lays the foundation for creating the impressive multithreaded programs that clients demand.

Chapter 15—Strings, Characters and Regular Expressions

In this chapter, we discuss the processing of words, sentences, characters and groups of characters. In C#, ***`string`***s (groups of characters) are objects. This is yet another benefit of C#'s emphasis on object-oriented programming. Objects of type **`string`** contain methods that can copy, create hash codes, search, extract substrings and concatenate strings with one another. As an interesting example of strings, we create a card shuffling-and-dealing simulation. We discuss regular expressions, a powerful tool for searching and manipulating text.

Chapter 16—Graphics and Multimedia

In this chapter, we discuss *GDI+* (an extension of the *Graphics Device Interface—GDI*), the Windows service that provides the graphical features used by .NET. The extensive graphical capabilities of GDI+ can make programs more visual and fun to create and use. We discuss C#'s treatment of graphics objects and color control, and we discuss how to draw arcs, polygons and other shapes. We use various pens and brushes to create color effects and include an example demonstrating gradient fills and textures. This chapter introduces techniques for turning text-only applications into exciting, aesthetically pleasing programs that even novice programmers can write with ease. The second half of the chapter focuses on audio, video and speech technology. We discuss adding sound, video and animated characters to programs (primarily using existing audio and video clips). You will see how easy it is to incorporate multimedia into C# applications. This chapter introduces an exciting technology called *Microsoft Agent* for adding *interactive animated characters* to a program. Each character allows users to interact with the application, using more natural human communication techniques, such as speech. The agent characters accept mouse and keyboard interaction, speak and hear (i.e., they support speech synthesis and speech recognition). With these capabilities, your applications can speak to users and actually respond to their voice commands!

Chapter 17—Files and Streams

Imagine a program that could not save data to a file. Once the program is closed, all the work performed in the program is lost forever. For this reason, this chapter is one of the most important for programmers who will be developing commercial applications. We explain how to input and output streams of data from and to files, respectively. We discuss how programs read and write data from and to secondary storage devices (such as disks). A detailed example demonstrates these concepts by allowing the user to read and write bank account information to and from files. We introduce those classes and methods in C# that help perform input and output conveniently—they demonstrate the power of object-oriented programming and reusable classes. We discuss benefits of sequential files, random-access files and buffering. This chapter is crucial for developing C# file-processing applications and networking applications (Chapter 22), which also use the techniques in this chapter to send and receive data.

Chapter 18—Extensible Markup Language (XML)[7]

The Extensible Markup Language (XML) derives from SGML (Standardized General Markup Language), which became an industry standard in 1986. Although SGML is employed in publishing applications worldwide, it has not been incorporated into the mainstream computing and information technology curricula because of its sheer size and complexity. XML is an effort to make SGML-like technology available to a much broader community. It was created by the World Wide Web Consortium (W3C) for describing data in a portable format, is one of most important technologies in industry today and is being integrated into almost every field. XML differs in concept from markup languages such as the HyperText Markup Language (HTML). HTML is a markup language for describing how information is rendered in a browser. XML is a language for creating markup languages for virtually any type of information. Document authors use XML to create entirely new markup languages to describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes. Markup languages created with XML include WML (Wireless Markup Language), XHTML (Extensible HyperText Markup Language, for Web content), MathML (for mathematics), VoiceXML™ (for speech), SMIL™ (Synchronized Multimedia Integration Language, for multimedia presentations), CML (Chemical Markup Language, for chemistry) and XBRL (Extensible Business Reporting Language, for financial data exchange). Companies and individuals constantly are finding new and exciting uses for XML. In this chapter, we present examples that illustrate the basics of marking up data with XML. We demonstrate several XML-derived markup languages, such as *XML Schema* (for checking an XML document's grammar), *XSLT (Extensible Stylesheet Language Transformations*, for transforming an XML document's data into another text-based format such as XHTML) and Microsoft's *BizTalk*™ (for marking up business transactions). (For readers who are unfamiliar with XHTML, we provide Appendices K and L, which provide a detailed introduction to XHTML.)

Chapter 19—Database, SQL and ADO .NET

Access and storage of data are integral to creating powerful software applications. This chapter discusses .NET support for database manipulation. Today's most popular database systems are relational databases. In this chapter, we introduce the Structured Query Language (SQL) for performing queries on relational databases. We introduce ADO .NET—an extension of Microsoft's ActiveX Data Objects that enables .NET applications to access and manipulate databases. ADO .NET allows data to be exported as XML, which enables applications that use ADO .NET to communicate with a variety of programs that understand XML. The reader will learn how to create database connections, using tools provided in Visual Studio .NET, and will learn how to use the classes in the **`System.Data`** namespace.

---

[7] The reader interested in a deeper treatment of XML may want to consider our book, *XML How to Program*.

Chapter 20—ASP .NET, Web Forms and Web Controls

Previous chapters demonstrated how to create applications that execute locally on the user's computer. In this chapter and the next, we discuss how to create Web-based applications using *Active Server Pages (ASP) .NET*. This is a crucial aspect of .NET and of Microsoft's vision of how software should be deployed on the Internet. ASP .NET is an integral technology for creating dynamic Web content marked up as HTML. (For readers who are unfamiliar with HTML, we provide a detailed introduction in Appendices I and J.) *Web Forms* provide GUIs for ASP .NET pages and can contain *Web controls*, such as labels, buttons and text boxes with which users interact. Like Windows Forms, Web Forms are designed using visual programming. This chapter presents many interesting examples, which include an online guest book application and a multi-tier, database intensive application that allows users to query a database for a list of publications by a specific author. Debugging Web Forms using the **`Trace`** property is also discussed.

Chapter 21—ASP .NET and Web Services

Chapter 21 continues our discussion of ASP .NET. In this chapter, we introduce *Web services*, which are programs that "expose" services (i.e., methods) to clients. Using Web services, programmers can create methods that anyone can invoke. This enables applications to invoke methods remotely over a network. Web services offer increased software reusability, making the Internet, in essence, a programming library available to programmers worldwide. Web services use XML and SOAP to mark up and send information, respectively. This chapter presents several interesting examples that include Web services for manipulating huge numbers (up to 100 digits), simulating the card game of blackjack and implementing an airline reservation system. One particularly interesting example is our temperature server, a Web service that gathers weather information for dozens of cities in the United States.

Chapter 22—Networking: Streams-Based Sockets and Datagrams

Chapter 22 introduces the fundamental techniques of C#-based networking—streams and datagrams. We demonstrate how streams-based *sockets* allow us to hide many networking details. With sockets, networking is as simple as if we were reading from and writing to a file. We also introduce *datagrams* in which packets of information are sent between programs. Each packet is addressed to its recipient and sent out to the network, which routes the packet to its destination. The examples in this chapter focus on communication between applications. One example demonstrates using streams-based sockets to communicate between two C# programs. Another similar example, sends datagrams between applications. We also show how to create a multithreaded-server application that can communicate multiple clients in parallel. In this client/server tic-tac-toe game, the server maintains the status of the game and two clients communicate with the server to play the game.

Chapter 23—Data Structures and Collections

This chapter discusses arranging data into aggregations such as linked lists, stacks, queues and trees. Each data structure has properties that are useful in a wide variety of applications, from sorting elements to keeping track of method calls. We discuss how to build each of these data structures. This is also a valuable experience in crafting useful classes. In addition, we cover prebuilt collection classes in the .NET Framework Class Library. These data structures have many useful methods for sorting, inserting, and deleting items, plus methods to enable data structures to resize themselves dynamically. When possible, C# programmers should use the Framework Class Library to find appropriate data structures, rather than implementing these data structures themselves. This chapter reinforces much of the object technology discussed in Chapters 8, 9 and 10, including classes, inheritance and composition.

Chapter 24—Accessibility

The World Wide Web presents a challenge to individuals with disabilities. Multimedia-rich Web sites are difficult for text readers and other programs to interpret; thus, users with hearing and visual impairments may have difficulty browsing such sites. To help rectify this situation, the World Wide Web Consortium (W3C) launched the *Web Accessibility Initiative (WAI)*, which provides guidelines for making Web sites accessible to people with disabilities. This chapter provides a description of these guidelines, such as the use of the **`<headers>`** tag to make tables more accessible to page readers, the **`alt`** attribute of the **`<img>`** tag to describe images, and XHTML and CSS to ensure that a page can be viewed on almost any type of display or reader. We illustrate key accessibility features of Visual Studio .NET and of Windows 2000. We also introduce *VoiceXML* and *CallXML*, two technologies for increasing the accessibility of Web-based content. VoiceXML helps people with visual impairments to access Web content via speech synthesis and speech recognition. CallXML allows users with visual impairments to access Web-based content through a telephone. In the chapter exercises, readers create their own voice mail applications, using CallXML.

Appendix A—Operator precedence chart

This appendix lists C# operators and their precedence.

Appendix B—Number Systems

This appendix explains the binary, octal, decimal and hexadecimal number systems. It also reviews the conversion of numbers among these bases and illustrates mathematical operations in each base.

Appendix C—Career Opportunities

This appendix provides career resources for C# programmers.

Appendix D—Visual Studio .NET Debugger

This appendix introduces the Visual Studio .NET debugger for locating logic errors in programs. Key features of this appendix include setting breakpoints, stepping through programs line-by-line and “watching” variable values.

Appendix E—Generating Documentation in Visual Studio

Appendix E discusses how to create comments within C# code that can be extracted to create powerful, XML-based documentation.

Appendix F—ASCII Character Set

This appendix contains a table of the 128 ASCII alphanumeric symbols and their corresponding ASCII (American Standard Code for Information Interchange) numbers.

Appendix G—Unicode®

This appendix introduces the Unicode Standard, an encoding scheme that assigns unique numeric values to the characters of most of the world's languages. We include a Windows application that uses Unicode encoding to print welcome messages in several different languages.

Appendix H—COM Integration

Prior to .NET, COM (Component Object Model) was critical for specifying how different Windows programming languages communicate at the binary level. For example, COM components such as ActiveX controls and ActiveX DLLs often were written in Microsoft Visual C++, but used in other programs. The .NET platform does not directly support COM components, but Microsoft provides tools for the integration of COM components with .NET applications. In this appendix, we explore some of these tools by integrating an ActiveX control and an ActiveX DLL into C# applications.

Appendices I and J—Introduction to HyperText Markup Language 4: 1 & 2 (on CD)

These appendices provide an introduction to *HTML*—the *Hypertext Markup Language*. HTML is a *markup language* for describing the elements of an HTML document (Web page) so that a browser, such as Microsoft's Internet Explorer, can render (i.e., display) that page. These appendices are included for our readers who do not know HTML or who would like a review of HTML before studying Chapter 20, ASP .NET, Web Forms and Web Controls. We do not present any C# programming in these appendices. Some key topics covered in Appendix I include: incorporating text and images in an HTML document, linking to other HTML documents on the Web, incorporating special characters (such as copyright and trademark symbols) into an HTML document and separating parts of an HTML document with horizontal lines (called *horizontal rules*). In Appendix J, we discuss more substantial HTML elements and features. We demonstrate how to present information in *lists* and *tables*. We discuss how to collect information from people browsing a site. We explain how to use *internal linking* and *image maps* to make Web pages easier to navigate. We also discuss how to use *frames* to display multiple documents in the browser window.

Appendices K and L—Introduction to XHTML: Parts 1 & 2 (on CD)

In these appendices, we introduce the Extensible Hypertext Markup Language (XHTML). XHTML is an emerging W3C technology designed to replace HTML as the primary means of describing Web content. As an XML-based language, XHTML is more robust and extensible than HTML. XHTML incorporates most of HTML 4's elements and attributes—the focus of these appendices. Appendices K and L are included for our readers who do not know XHTML or who would like a review of XHTML before studying Chapter 18, Extensible Markup Language (XML) and Chapter 24, Accessibility.

Appendix M—HTML/XHTML Special Characters (on CD)

This appendix provides many commonly used HTML/XHTML special characters, called *character entity references*.

Appendix N—HTML/XHTML Colors (on CD)

This appendix lists commonly used HTML/XHTML color names and their corresponding hexadecimal values.

Appendix O—Bit Manipulation
This appendix discusses C#'s powerful bit-manipulation capabilities. This helps programs process bit strings, set individual bits on or off and store information more compactly. Such capabilities—inherited from C—are characteristic of low-level assembly languages and are valued by programmers writing systems software, such as operating system and networking software.

1.19 Internet and World Wide Web Resources

`www.deitel.com`
This is the official Deitel & Associates, Inc. Web site. Here you will find updates, corrections, downloads and additional resources for all Deitel publications. In addition, this site provides information about Deitel & Associates, Inc. professional, on-site seminars offered worldwide. In the near future, you will be able to register here to receive the *Deitel Buzz* e-mail newsletter.

`www.prenhall.com/deitel`
This is Prentice Hall's Web site for Deitel publications, which contains information about our products and publications, downloads, Deitel curriculum and author information.

`www.InformIT.com/deitel`
This is the Deitel & Associates, Inc. page on the InformIT Web site. InformIT is an all-around resource for IT professionals providing articles, electronic publications and other resources for today's hottest technologies. The Deitel kiosk at InformIT.com will have free articles and for-purchase electronic publications. In addition, you can purchase all Deitel products at this site.

`www.w3.org`
The World Wide Web Consortium (W3C) is an organization that develops and recommends technologies for the Internet and World Wide Web. This site includes links to W3C technologies, news, mission statements and frequently asked questions (FAQs).

`www.elsop.com/wrc/h_comput.htm`
This site contains presents the history of computing, content about famous innovators, the evolution of languages and the development of operating systems.

`www.w3.org/History.html`
This site overviews the history of the Internet. After briefly covering developments from 1945 –1988, the site details technological advances on a year-by-year basis, from 1989 to the present day.

`www.netvalley.com/intval.html`
This site presents the history of the Internet and the World Wide Web.

`www.microsoft.com`
The Microsoft Corporation Web site provides information and technical resources for all Microsoft products, including .NET, enterprise software and the Windows operating system.

SUMMARY

[This chapter is primarily a summary of the rest of the book, so we have not provided a summary section. The remaining chapters include detailed summaries of their contents.]

TERMINOLOGY

action
"administrative" section of the computer
Advanced Research Projects Agency (ARPA)
algorithm
Apple Computer
arithmetic and logic unit (ALU)

assembler
assembly language
bandwidth
batch
batch processing
building-block approach
C programming language
C# programming language
C++ programming language
calculation
Cascading Style Sheets (CSS)
central processing unit (CPU)
clarity
class
class libraries
Common Language Runtime (CLR)
Common Language Specification (CLS)
compiler
component
computation
computer
computer program
computer programmer
data
data independence
decision
disk
distributed computing
ECMA (European Computer Manufacturer’s Association)
e-mail (electronic mail)
Framework Class Library (FCL)
functionalization
HailStorm Web service
hardware
hardware platform
high-level language
HTML (HyperText Markup Language)
HTTP (HyperText Transfer Protocol)
IBM (International Business Machines)
input device
input unit
Internet
interpreter
intranet
IP (Internet Protocol)
Java programming language
job
keyboard
language independence
language interoperability
live-code™ approach
logical decision
logical unit
machine dependent
machine language
maintenance of software
“manufacturing” section of the computer
memory
memory unit
Microsoft .NET
Microsoft Intermediate Language (MSIL)
mouse
multiprogramming
multitasking
.NET Framework
.NET initiative
.NET language
n-tier application
object
object-based programming
object-oriented language
object-oriented programming (OOP)
operating system
output device
output unit
Pascal programming language
personal computer
platform independence
portability
porting
primary memory
processing unit
program
programmer
property of an object
“receiving” section of the computer
reusable software component
screen
secondary storage
share the resources of a computer
“shipping” section of the computer
silicon chip
SOAP (Simple Object Access Protocol)
software
software component
software reuse
structured programming
subscription-based software
task

TCP (Transmission Control Protocol)
TCP/IP (Transmission Control Protocol/Internet Protocol)
terminal
throughput
timesharing
translator program
universal data access
UNIX
virtual-memory operating system
Visual Basic .NET programming language
visual programming
W3C (World Wide Web Consortium)
W3C Recommendation
"warehouse" section of the computer
Web Form
Web service
Web site
Win32 API (Windows 32-bit Application Programming Interface)
World Wide Web (WWW)
XML (Extensible Markup Language)

SELF-REVIEW EXERCISES

1.1 Fill in the blanks in each of the following statements:

a) A computer can directly understand only its native __________ language, which is composed only of 1s and 0s.
b) Computers process data under the control of sets of instructions called computer __________.
c) SOAP is an acronym for __________.
d) __________ is a technology derived from SGML that is used to create mark up languages.
e) The three types of languages discussed in the chapter are machine languages, __________ and __________.
f) Programs that translate high-level language programs into machine language are called __________.
g) Visual Studio .NET is a/an __________ (IDE) in which C# programs are developed.
h) C is widely known as the development language of the __________ operating system.
i) The __________ provides a large programming library for .NET languages.
j) The Department of Defense developed the Ada language with a capability called multitasking, which allows programmers to specify activities that can proceed in parallel. C# offers a similar capability called __________.
k) Web services use __________ and __________ to mark up and send information over the Internet, respectively.

1.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Universal data access is an essential part of .NET.
b) W3C standards are called recommendations.
c) C# is an object-oriented language.
d) The Common Language Runtime (CLR) requires that programmers manage their own memory.
e) C# is the only language available for programming .NET applications.
f) Procedural programming models the world more naturally than object-oriented programming.
g) Computers can directly understand high-level languages.
h) MSIL is the common intermediate format to which all .NET programs compile, regardless of their original .NET language.
i) The .NET Framework is portable to non-Windows platforms.
j) Compiled programs run faster than their corresponding interpreted programs.
k) Throughput is the amount of work a computer can process in a given time period.

ANSWERS TO SELF-REVIEW EXERCISES

1.1 a) machine. b) programs. c) Simple Object Access Protocol. d) XML. e) assembly languages, high-level languages. f) compilers. g) integrated development environment (IDE). h) UNIX. i) Framework Class Library (FCL). j) multithreading. k) XML, SOAP.

1.2 a) True. b) True. c) True. d) False. The CLR handles memory management. e) False. C# is one of many .NET languages (others include Visual Basic and Visual C++). f) False. Object-oriented programming is a more natural way to model the world than is procedural programming. g) False. Computers can directly understand only their own machine languages. h) True. i) True. j) True. k) True.

EXERCISES

1.3 Categorize each of the following items as either hardware or software:

a) CPU.
b) Compiler.
c) Input unit.
d) A word-processor program.
e) A Visual Basic .NET program.

1.4 Distinguish between the terms HTML, XML and XHTML.

1.5 Translator programs, such as assemblers and compilers, convert programs from one language (referred to as the source language) to another language (referred to as the object language or target language). Determine which of the following statements are *true* and which are *false*:

a) An assembler translates source language programs into machine language programs.
b) A compiler converts source-language programs into object-language programs.
c) High-level languages are generally machine dependent.
d) A machine-language program requires translation before it can be run on a computer.
e) The Visual Basic .NET compiler translates a high-level language into SMIL.

1.6 What are the basic requirements of a .NET language? What is needed to run a .NET program on a new type of computer (machine)?

1.7 Expand each of the following acronyms:

a) W3C.
b) XML.
c) SOAP.
d) TCP/IP.
e) OOP.
f) CLR.
g) CLS.
h) FCL.
i) MSIL.

1.8 What are the key benefits of the .NET Framework and the CLR? What are the drawbacks?

1

Introduction to Computers, the Internet, the Web and C#

Objectives

- To understand basic computer concepts.
- To learn about various programming languages.
- To become familiar with the history of the C# programming language.
- To understand the Microsoft® .NET initiative.
- To preview the remaining chapters of the book.

Things are always at their best in their beginning.
Blaise Pascal

High thoughts must have high language.
Aristophanes

Our life is frittered away by detail…Simplify, simplify.
Henry David Thoreau

Before beginning, plan carefully….
Marcus Tullius Cicero

Look with favor upon a bold beginning.
Virgil

I think I'm beginning to learn something about it.
Auguste Renoir

Outline

1.1 Introduction
1.2 What Is a Computer?
1.3 Computer Organization
1.4 Evolution of Operating Systems
1.5 Personal Computing, Distributed Computing and Client/Server Computing
1.6 Machine Languages, Assembly Languages and High-level Languages
1.8 C#
1.7 C, C++, Visual Basic .NET and Java™
1.9 Other High-level Languages
1.10 Structured Programming
1.11 Key Software Trend: Object Technology
1.12 Hardware Trends
1.13 History of the Internet and World Wide Web
1.14 World Wide Web Consortium (W3C)
1.15 Extensible Markup Language (XML)
1.16 Introduction to Microsoft .NET
1.17 .NET Framework and the Common Language Runtime
1.18 Tour of the Book
1.19 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

1.1 Introduction

Welcome to C#! In creating this book, we have worked hard to provide students with the most accurate and complete information regarding the C# language, and the .NET platform. The book is designed to be appropriate for readers at all levels, from practicing programmers to individuals with little or no programming experience. We hope that working with this text will be an informative, entertaining and challenging learning experience for you.

How can one book appeal to both novices and skilled programmers? The core of this book emphasizes the achievement of program clarity through proven techniques of *structured programming*, *object-based programming*, *object-oriented programming (OOP)* and *event-driven programming*. Nonprogrammers learn basic skills that underlie good programming; experienced developers receive a rigorous explanation of the language and may improve their programming styles. Perhaps most importantly, the book presents hundreds of complete, working C# programs and depicts their outputs. We call this the *LIVE-CODE™ approach*. All of the book's examples are available on the CD-ROM that accompanies this book and on our Web site, **`www.deitel.com`**.

Computer use is increasing in almost every field of endeavor. In an era of steadily rising costs, computing costs have decreased dramatically because of rapid developments in both hardware and software technology. Computers that filled large rooms and cost millions of dollars just two decades ago now can be inscribed on the surfaces of silicon chips smaller than a fingernail, costing perhaps a few dollars each. Silicon is one of the most abundant materials on earth—it is an ingredient in common sand. Silicon-chip technology has made computing so economical that hundreds of millions of general-purpose computers are in use worldwide, helping people in business, industry, government and their personal lives. Given the current rate of technological development, this number could easily double over the next few years.

In beginning to study this text, you are starting on a challenging and rewarding educational path. As you proceed, if you would like to communicate with us, please send an e-mail to **`deitel@deitel.com`** or browse our World Wide Web sites at **`www.deitel.com`**, **`www.prenhall.com/deitel`** and **`www.InformIT.com/deitel`**. We hope that you enjoy learning C# through reading *C# How to Program*.

1.2 What Is a Computer?

A *computer* is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than those of human beings. For example, many of today’s personal computers can perform hundreds of millions—even billions—of additions per second. A person operating a desk calculator might require decades to complete the same number of calculations that a powerful personal computer can perform in one second. (*Points to ponder*: How would you know whether the person had added the numbers correctly? How would you know whether the computer had added the numbers correctly?) Today’s fastest *supercomputers* can perform hundreds of billions of additions per second—about as many calculations as hundreds of thousands of people could perform in one year! Trillion-instruction-per-second computers are already functioning in research laboratories!

Computers process *data* under the control of sets of instructions called *computer programs*. These programs guide computers through orderly sets of actions that are specified by individuals known as *computer programmers*.

A computer is composed of various devices (such as the keyboard, screen, mouse, disks, memory, CD-ROM and processing units) known as *hardware*. The programs that run on a computer are referred to as *software*. Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. Software-development costs, however, have been rising steadily, as programmers develop ever more powerful and complex applications without being able to improve significantly the technology of software development. In this book, you will learn proven software-development methods that can reduce software-development costs—top-down stepwise refinement, functionalization and object-oriented programming. Object-oriented programming is widely believed to be the significant breakthrough that can greatly enhance programmer productivity.

1.3 Computer Organization

Virtually every computer, regardless of differences in physical appearance, can be envisioned as being divided into six *logical units*, or sections:

1. *Input unit.* This “receiving” section of the computer obtains information (data and computer programs) from various *input devices*. The input unit then places this information at the disposal of the other units to facilitate the processing of the information. Today, most users enter information into computers via keyboards and mouse devices. Other input devices include microphones (for speaking to the computer), scanners (for scanning images) and digital cameras (for taking photographs and making videos).
2. *Output unit.* This “shipping” section of the computer takes information that the computer has processed and places it on various *output devices*, making the information available for use outside the computer. Computers can output information in various ways, including displaying the output on screens, playing it on audio/video devices, printing it on paper or using the output to control other devices.
3. *Memory unit.* This is the rapid-access, relatively low-capacity “warehouse” section of the computer, which facilitates the temporary storage of data. The memory unit retains information that has been entered through the input unit, enabling that information to be immediately available for processing. In addition, the unit retains processed information until that information can be transmitted to output devices. Often, the memory unit is called either *memory* or *primary memory*—*random access memory* (*RAM*) is an example of primary memory. Primary memory is usually volatile, which means that it is erased when the machine is powered off.
4. *Arithmetic and logic unit* (*ALU*). The ALU is the “manufacturing” section of the computer. It is responsible for the performance of calculations such as addition, subtraction, multiplication and division. It also contains decision mechanisms, allowing the computer to perform such tasks as determining whether two items stored in memory are equal.
5. *Central processing unit* (*CPU*). The CPU serves as the “administrative” section of the computer. This is the computer’s coordinator, responsible for supervising the operation of the other sections. The CPU alerts the input unit when information should be read into the memory unit, instructs the ALU about when to use information from the memory unit in calculations and tells the output unit when to send information from the memory unit to certain output devices.
6. *Secondary storage unit.* This unit is the long-term, high-capacity “warehousing” section of the computer. Secondary storage devices, such as hard drives and disks, normally hold programs or data that other units are not actively using; the computer then can retrieve this information when it is needed—hours, days, months or even years later. Information in secondary storage takes much longer to access than does information in primary memory. However, the price per unit of secondary storage is much less than the price per unit of primary memory. Secondary storage is usually *nonvolatile*—it retains information even when the computer is off.

1.4 Evolution of Operating Systems

Early computers were capable of performing only one *job* or *task* at a time. In this mode of computer operation, often called single-user *batch processing*, the computer runs one program at a time and processes data in groups called *batches*. Users of these early systems typically submitted their jobs to a computer center on decks of punched cards. Often, hours or even days elapsed before results were returned to the users' desks.

To make computer use more convenient, software systems called *operating systems* were developed. Early operating systems oversaw and managed computers' transitions between jobs. By minimizing the time it took for a computer operator to switch from one job to another, the operating system increased the total amount of work, or *throughput*, computers could process in a given time period.

As computers became more powerful, single-user batch processing became inefficient, because computers spent a great deal of time waiting for slow input/output devices to complete their tasks. Developers then looked to multiprogramming techniques, which enabled many tasks to *share* the resources of the computer to achieve better utilization. *Multiprogramming* involves the "simultaneous" operation of many jobs on a computer that splits its resources among those jobs. However, users of early multiprogramming operating systems still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several industry and university groups pioneered *timesharing* operating systems. Timesharing is a special type of multiprogramming that allows users to access a computer through *terminals* (devices with keyboards and screens). Dozens or even hundreds of people can use a timesharing computer system at once. It is important to note that the computer does not actually run all the users' requests simultaneously. Rather, it performs a small portion of one user's job and moves on to service the next user. However, because the computer does this so quickly, it can provide service to each user several times per second. This gives users' programs the appearance of running simultaneously. Timesharing offers major advantages over previous computing systems in that users receive prompt responses to requests, instead of waiting long periods to obtain results.

The UNIX operating system, which is now widely used for advanced computing, originated as an experimental timesharing operating system. Dennis Ritchie and Ken Thompson developed UNIX at Bell Laboratories beginning in the late 1960s and developed C as the language in which they wrote it. They freely distributed the source code to other programmers who wanted to use, modify and extend it. A large community of UNIX users quickly developed. The operating system grew as UNIX users contributed their own programs and tools. Through a collaborative effort among numerous researchers and developers, UNIX became a powerful and flexible operating system able to handle almost any type of task that a user required. Many versions of UNIX have evolved, including today's phenomenally popular *open-source* Linux operating system. Typically, the source code for open-source products is freely available over the Internet. This enables developers to learn from, validate and modify the source code. Often, open-source products require that developers publish any enhancements they make so the open-source community can continue to evolve those products.

1.5 Personal Computing, Distributed Computing and Client/Server Computing

In 1977, Apple Computer popularized the phenomenon of *personal computing*. Initially, it was a hobbyist's dream. However, the price of computers soon dropped so far that large numbers of people could buy them for personal or business use. In 1981, IBM, the world's largest computer vendor, introduced the IBM Personal Computer. Personal computing rapidly became legitimate in business, industry and government organizations.

The computers first pioneered by Apple and IBM were “stand-alone” units—people did their work on their own machines and transported disks back and forth to share information. (This process was often called “sneakernet.”) Although early personal computers were not powerful enough to timeshare several users, the machines could be linked together into computer networks, either over telephone lines or via *local area networks* (*LANs*) within an organization. These networks led to the *distributed computing* phenomenon, in which an organization’s computing is distributed over networks to the sites at which the work of the organization is performed, instead of being performed only at a central computer installation. Personal computers were powerful enough to handle both the computing requirements of individual users and the basic tasks involved in the electronic transfer of information between computers. *N-tier applications* split up an application over numerous distributed computers. For example, a *three-tier application* might have a user interface on one computer, business-logic processing on a second and a database on a third; all interact as the application runs.

Today’s most advanced personal computers are as powerful as the million-dollar machines of just two decades ago. High-powered desktop machines—called *workstations*—provide individual users with enormous capabilities. Information is easily shared across computer networks, in which computers called *servers* store programs and data that can be used by *client* computers distributed throughout the network. This type of configuration gave rise to the term *client/server computing*. Today’s popular operating systems, such as UNIX, Solaris, MacOS, Windows 2000, Windows XP and Linux, provide the kinds of capabilities discussed in this section.

1.6 Machine Languages, Assembly Languages and High-level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others that require intermediate *translation* steps. Although hundreds of computer languages are in use today, the diverse offerings can be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can understand only its own *machine language* directly. As the “natural language” of a particular computer, machine language is defined by the computer’s hardware design. Machine languages generally consist of streams of numbers (ultimately reduced to 1s and 0s) that instruct computers how to perform their most elementary operations. Machine languages are *machine-dependent*, which means that a particular machine language can be used on only one type of computer. The following section of a machine-language program, which adds *overtime* pay to *base pay* and stores the result in *gross pay*, demonstrates the incomprehensibility of machine language to the human reader.

```
+1300042774
+1400593419
+1200274027
```

As the popularity of computers increased, machine-language programming proved to be excessively slow, tedious and error prone. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the elementary operations of the computer. These abbreviations formed the basis of *assembly languages*. *Translator programs* called *assemblers* convert assembly language programs to machine language at computer speeds. The following section of an assembly-language program also adds *overtime pay* to *base pay* and stores the result in *gross pay*, but presents the steps more clearly to human readers than does its machine-language equivalent:

```
LOAD   BASEPAY
ADD    OVERPAY
STORE  GROSSPAY
```

Such code is clearer to humans but incomprehensible to computers until translated into machine language.

Although computer use increased rapidly with the advent of assembly languages, these languages still required many instructions to accomplish even the simplest tasks. To speed up the programming process, *high-level languages*, in which single statements accomplish substantial tasks, were developed. Translation programs called *compilers* convert high-level-language programs into machine language. High-level languages enable programmers to write instructions that look almost like everyday English and contain common mathematical notations. A payroll program written in a high-level language might contain a statement such as

```
grossPay = basePay + overTimePay
```

Obviously, programmers prefer high-level languages to either machine languages or assembly languages.

The compilation of a high-level language program into machine language can require a considerable amount of time. However, this problem was solved by the development of *interpreter* programs that can execute high-level language programs directly, bypassing the compilation step. Although programs that are already compiled execute faster than interpreted programs, interpreters are popular in program-development environments. In these environments, developers change programs frequently as they add new features and correct errors. Once a program is fully developed, a compiled version can be produced so that the program runs at maximum efficiency.

1.7 C, C++, Visual Basic .NET and Java™

As high-level languages develop, new offerings build on aspects of their predecessors. C++ evolved from C, which in turn evolved from two previous languages, BCPL and B. Martin Richards developed BCPL in 1967 as a language for writing operating systems, software and compilers. Ken Thompson modeled his language, B, after BCPL. In 1970, Thompson used B to create early versions of the UNIX operating system. Both BCPL and B were "typeless" languages, meaning that every data item occupied one "word" in memory. Using these languages, programmers assumed responsibility for treating each data item as a whole number or real number, for example.

The C language, which Dennis Ritchie evolved from B at Bell Laboratories, was originally implemented in 1973. Although C employs many of BCPL and B's important concepts, it also offers data typing and other features. C first gained widespread recognition as a development language of the UNIX operating system. However, C is now available for most computers, and many of today’s major operating systems are written in C or C++. C is a hardware-independent language, and, with careful design, it is possible to write C programs that are portable to most computers.

C++, an extension of C using elements from Simula 67, a simulation programming language, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that “spruce up” the C language, but, more importantly, it provides capabilities for *object-oriented programming (OOP)*.

At a time when demand for new and more powerful software is soaring, the ability to build software quickly, correctly and economically remains an elusive goal. However, this problem can be addressed in part through the use of *objects*, or reusable software *components* that model items in the real world (see Section 1.11). Software developers are discovering that a modular, object-oriented approach to design and implementation can make software development groups much more productive than is possible via previous popular programming techniques, such as structured programming. Furthermore, object-oriented programs are often easier to understand, correct and modify.

In addition to C++, many other object-oriented languages have been developed. These include Smalltalk, which was created at Xerox's Palo Alto Research Center (PARC). Smalltalk is a pure object-oriented language, which means that literally everything is an object. C++ is a hybrid language—it is possible to program in a C-like style, an object-oriented style or both. Although some perceive this range of options as a benefit, most programmers today believe that it is best to program in a purely object-oriented manner.

Developing Microsoft Windows-based applications in languages such as C and C++, however, proved to be a difficult and cumbersome process. When Bill Gates founded Microsoft Corporation, he implemented *BASIC* on several early personal computers. BASIC (Beginner’s All-Purpose Symbolic Instruction Code) is a programming language developed in the mid-1960s by Professors John Kemeny and Thomas Kurtz of Dartmouth College as a language for writing simple programs. BASIC’s primary purpose was to familiarize novices with programming techniques. The natural evolution from BASIC to Visual Basic was introduced in 1991 as a result of the development of the Microsoft Windows graphical user interface (GUI) in the late 1980s and the early 1990s.

Although Visual Basic is derived from the BASIC programming language, it is a distinctly different language that offers such powerful features as graphical user interfaces, event handling, access to the *Windows 32-bit Application Programming Interface* (*Win32 API*), object-oriented programming and error handling. Visual Basic is one of the most popular event-driven, visual programming interfaces.

The latest version of Visual Basic, called *Visual Basic .NET*[1], is designed for Microsoft’s new programming platform, .NET. Earlier versions of Visual Basic provided object-oriented capabilities, but Visual Basic .NET offers enhanced object orientation and makes use of the powerful library of reusable software components in .NET.

Around the same time that Visual Basic was being developed, many individuals projected that intelligent consumer-electronic devices would be the next major market in which microprocessors would have a profound impact. Recognizing this, Sun Microsystems in 1991 funded an internal corporate research project code-named Green. The project

---

1. The reader interested in Visual Basic .NET may want to consider our book, *Visual Basic .NET How to Program, Second Edition.*

resulted in the development of a language based on C and C++. Although the language's creator, James Gosling, called it Oak (after an oak tree outside his window at Sun), it was later discovered that a computer language called Oak already existed. When a group of Sun employees visited a local coffee place, the name Java was suggested, and it stuck.

Unfortunately, the Green project ran into some difficulties. The marketplace for intelligent consumer-electronic devices was not developing as quickly as Sun had anticipated. Worse yet, a major contract for which Sun competed was awarded to another company. The project was, at this point, in danger of being canceled. By sheer good fortune, the World Wide Web exploded in popularity in 1993, and Sun saw immediate potential for using Java to design *dynamic content* (i.e., animated and interactive content) for Web pages.

Sun formally announced Java at a conference in May 1995. Ordinarily, an event like this would not generate much publicity. However, Java grabbed the immediate attention of the business community because of the new, widespread interest in the World Wide Web. Developers now use Java to create Web pages with dynamic content, to build large-scale enterprise applications, to enhance the functionality of World Wide Web servers (the computers that provide the content distributed to our Web browsers when we browse Web sites), to provide applications for consumer devices (e.g., cell phones, pagers and PDAs) and for many other purposes.

1.8 C#

The advancement of programming tools (e.g., C++ and Java) and consumer-electronic devices (e.g., cell phones) created problems and new requirements. The integration of software components from various languages proved difficult, and installation problems were common because new versions of shared components were incompatible with old software. Developers also discovered they needed Web-based applications that could be accessed and used via the Internet. As a result of mobile electronic device popularity, software developers realized that their clients were no longer restricted to desktop computers. Developers recognized the need for software that was accessible to anyone and available via almost any type of device. To address these needs, Microsoft announced its *.NET* (pronounced "dot-net") *initiative* and the *C#* (pronounced "C-Sharp") programming language.

The *.NET platform* is one over which Web-based applications can be distributed to a great variety of devices (even cell phones) and to desktop computers. The platform offers a new software-development model that allows applications created in disparate programming languages to communicate with each other. The C# programming language, developed at Microsoft by a team led by Anders Hejlsberg and Scott Wiltamuth, was designed specifically for the .NET platform as a language that would enable programmers to migrate easily to .NET. This migration is made easy due to the fact that C# has roots in C, C++ and Java, adapting the best features of each and adding new features of its own. Because C# has been built upon such widely used and well-developed languages, programmers will find learning C# to be easy and enjoyable.

C# is an event-driven, fully object-oriented, visual programming language in which programs are created using an *Integrated Development Environment* (*IDE*). With the IDE, a programmer can create, run, test and debug C# programs conveniently, thereby reducing the time it takes to produce a working program to a fraction of the time it would have taken without using the IDE. The process of rapidly creating an application using an IDE is typically referred to as *Rapid Application Development (RAD)*.

C# also enables a new degree of language interoperability: Software components from different languages can interact as never before. Developers can package even old software to work with new C# programs. In addition, C# applications can interact via the Internet, using industry standards such as the Simple Object Access Protocol (SOAP) and XML, which we discuss in Chapter 18, Extensible Markup Language (XML). The programming advances embodied in .NET and C# will lead to a new style of programming, in which applications are created from building blocks available over the Internet.

1.9 Other High-level Languages

Although hundreds of high-level languages have been developed, only a few have achieved broad acceptance. This section overviews several languages that, like BASIC, are long-standing and popular high-level languages. IBM Corporation developed Fortran (FORmula TRANslator) between 1954 and 1957 to create scientific and engineering applications that require complex mathematical computations. Fortran is still widely used.

COBOL (COmmon Business Oriented Language) was developed in 1959 by a group of computer manufacturers in conjunction with government and industrial computer users. COBOL is used primarily for commercial applications that require the precise and efficient manipulation of large amounts of data. A considerable portion of today's business software is still programmed in COBOL. Approximately one million programmers are actively writing in COBOL.

Pascal was designed in the late 1960s by Professor Nicklaus Wirth and was intended for academic use. We explore Pascal in the next section.

1.10 Structured Programming

During the 1960s, many large software-development efforts encountered severe difficulties. Development typically ran behind schedule, costs greatly exceeded budgets and the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity, intended to address these issues, resulted in the evolution of *structured programming*—a disciplined approach to the creation of programs that are clear, demonstrably correct and easy to modify.

One of the more tangible results of this research was the development of the *Pascal* programming language in 1971. Pascal, named after the seventeenth-century mathematician and philosopher Blaise Pascal, was designed for teaching structured programming in academic environments and rapidly became the preferred introductory programming language in most universities. Unfortunately, because the language lacked many features needed to make it useful in commercial, industrial and government applications, it was not widely accepted in these environments. By contrast, C, which also arose from research on structured programming, did not have the limitations of Pascal, and programmers quickly adopted it.

The *Ada* programming language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of programming languages were being used to produce DOD's massive command-and-control software systems. DOD wanted a single language that would meet its needs. Pascal was chosen as a base, but the final Ada language is quite different from Pascal. The language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is generally credited with writing the world’s first computer program, in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada is *multitasking*, which allows programmers to specify that many activities are to occur in parallel. As we will see in Chapter 14, C# offers a similar capability, called *multithreading*.

1.11 Key Software Trend: Object Technology

One of the authors, HMD, remembers the great frustration felt in the 1960s by software-development organizations, especially those developing large-scale projects. During the summers of his undergraduate years, HMD had the privilege of working at a leading computer vendor on the teams developing time-sharing, virtual-memory operating systems. It was a great experience for a college student, but, in the summer of 1967, reality set in. The company “decommitted” from producing as a commercial product the particular system that hundreds of people had been working on for several years. It was difficult to get this software right. Software is “complex stuff.”

As the benefits of structured programming (and the related disciplines of *structured systems analysis and design*) were realized in the 1970s, improved software technology did begin to appear. However, it was not until the technology of object-oriented programming became widely used in the 1980s and 1990s that software developers finally felt they had the necessary tools to improve the software-development process dramatically.

Actually, object technology dates back to at least the mid-1960s, but no broad-based programming language incorporated the technology until C++. Although not strictly an object-oriented language, C++ absorbed the capabilities of C and incorporated Simula’s ability to create and manipulate objects. C++ was never intended for widespread use beyond the research laboratories at AT&T, but grass-roots support rapidly developed for the hybrid language.

What are objects, and why are they special? Object technology is a packaging scheme that facilitates the creation of meaningful software units. These units are large and focused on particular applications areas. There are date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects and so on. In fact, almost any noun can be reasonably represented as a software object. Objects have *properties* (i.e., *attributes*, such as color, size and weight) and perform *actions* (i.e., *behaviors*, such as moving, sleeping or drawing). Classes represent groups of related objects. For example, all cars belong to the “car” class, even though individual cars vary in make, model, color and options packages. A class specifies the general format of its objects; the properties and actions available to an object depend on its class.

We live in a world of objects. Just look around you—there are cars, planes, people, animals, buildings, traffic lights, elevators and so on. Before object-oriented languages appeared, *procedural programming languages* (such as Fortran, Pascal, BASIC and C) focused on actions (verbs) rather than things or objects (nouns). We live in a world of objects, but earlier programming languages forced individuals to program primarily with verbs. This paradigm shift made program writing a bit awkward. However, with the advent of popular object-oriented languages, such as C++, Java and C#, programmers can program in an object-oriented manner that reflects the way in which they perceive the world. This process, which seems more natural than procedural programming, has resulted in significant productivity gains.

One of the key problems with procedural programming is that the program units created do not mirror real-world entities effectively and therefore are not particularly reusable. Programmers often write and rewrite similar software for various projects. This wastes precious time and money as people repeatedly “reinvent the wheel.” With object technology, properly designed software entities (called objects) can be reused on future projects. Using libraries of reusable componentry can greatly reduce the amount of effort required to implement certain kinds of systems (as compared to the effort that would be required to reinvent these capabilities in new projects). C# programmers use the .NET Framework Class Library (known commonly as the FCL).

Some organizations report that software reusability is not, in fact, the key benefit of object-oriented programming. Rather, they indicate that object-oriented programming tends to produce software that is more understandable because it is better organized and has fewer maintenance requirements. As much as 80 percent of software costs are not associated with the original efforts to develop the software, but instead are related to the continued evolution and maintenance of that software throughout its lifetime. Object orientation allows programmers to abstract the details of software and focus on the “big picture.” Rather than worrying about minute details, the programmer can focus on the behaviors and interactions of objects. A roadmap that showed every tree, house and driveway would be difficult, if not impossible, to read. When such details are removed and only the essential information (roads) remains, the map becomes easier to understand. In the same way, a program that is divided into objects is easy to understand, modify and update because it hides much of the detail. It is clear that object-oriented programming will be the key programming methodology for at least the next decade.

Software Engineering Observation 1.1

Use a building-block approach to creating programs. By using existing pieces in new projects, programmers avoid reinventing the wheel. This is called software reuse, *and it is central to object-oriented programming.*

[*Note*: We will include many of these *Software Engineering Observations* throughout the text to explain concepts that affect and improve the overall architecture and quality of a software system and, particularly, of large software systems. We will also highlight *Good Programming Practices* (practices that can help you write programs that are clearer, more understandable, more maintainable and easier to test and debug), *Common Programming Errors* (problems to watch for to ensure that you do not make these same errors in your programs), *Performance Tips* (techniques that will help you write programs that run faster and use less memory), *Portability Tips* (techniques that will help you write programs that can run, with little or no modification, on a variety of computers), *Testing and Debugging Tips* (techniques that will help you remove bugs from your programs and, more importantly, write bug-free programs in the first place) and *Look-and-Feel Observations* (techniques that will help you design the “look and feel” of your graphical user interfaces for appearance and ease of use). Many of these techniques and practices are only guidelines; you will, no doubt, develop your own preferred programming style.]

The advantage of creating your own code is that you will know exactly how it works. The code will be yours to examine, modify and improve. The disadvantage is the time and effort that goes into designing, developing and testing new code.

Performance Tip 1.1

Reusing proven code components instead of writing your own versions can improve program performance, because these components normally are written to perform efficiently.

Software Engineering Observation 1.2

Extensive class libraries of reusable software components are available over the Internet and the World Wide Web; many are offered free of charge.

1.12 Hardware Trends

Every year, people generally expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the costs of hardware supporting these technologies. For many decades, and continuing into the foreseeable future, hardware costs have fallen rapidly, if not precipitously. Every year or two, the capacities of computers approximately double.[2] This is especially true in relation to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) computers have to hold programs and data over longer periods of time and their processor speeds—the speeds at which computers execute their programs (i.e., do their work). Similar improvements have occurred in the communications field, in which costs have plummeted as enormous demand for communications bandwidth (i.e., information-carrying capacity) has attracted tremendous competition. We know of no other fields in which technology moves so quickly and costs fall so rapidly. Such phenomenal improvement in the computing and communications fields is truly fostering the so-called *Information Revolution*.

When computer use exploded in the 1960s and 1970s, many discussed the dramatic improvements in human productivity that computing and communications would cause. However, these improvements did not materialize. Organizations were spending vast sums of capital on computers and employing them effectively, but without fully realizing the expected productivity gains. The invention of microprocessor chip technology and its wide deployment in the late 1970s and 1980s laid the groundwork for the productivity improvements that individuals and businesses have achieved in recent years.

1.13 History of the Internet and World Wide Web

In the late 1960s, one of the authors (HMD) was a graduate student at MIT. His research at MIT's Project Mac (now the Laboratory for Computer Science—the home of the World Wide Web Consortium) was funded by ARPA—the Advanced Research Projects Agency of the Department of Defense. ARPA sponsored a conference at which several dozen ARPA-funded graduate students were brought together at the University of Illinois at Urbana-Champaign to meet and share ideas. During this conference, ARPA rolled out the blueprints for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at a then-stunning 56 Kbps (1 Kbps is equal to 1,024 bits per second), at a time when most people (of the few who had access to networking technologies) were connecting over telephone lines to computers at a rate of 110 bits per second.

---

[2] This often is called *Moore's Law*.

HMD vividly recalls the excitement at that conference. Researchers at Harvard talked about communicating with the Univac 1108 "supercomputer," which was located across the country at the University of Utah, to handle calculations related to their computer graphics research. Many other intriguing possibilities were discussed. Academic research was about to take a giant leap forward. Shortly after this conference, ARPA proceeded to implement what quickly became called the *ARPAnet*, the grandparent of today's *Internet*.

Things worked out differently from the original plan. Although the ARPAnet did enable researchers to network their computers, its chief benefit proved to be the capability for quick and easy communication via what came to be known as *electronic mail* (*e-mail*). This is true even on today's Internet, with e-mail, instant messaging and file transfer facilitating communications among hundreds of millions of people worldwide.

The network was designed to operate without centralized control. This meant that, if a portion of the network should fail, the remaining working portions would still be able to route data packets from senders to receivers over alternative paths.

The protocol (i.e., set of rules) for communicating over the ARPAnet became known as the *Transmission Control Protocol (TCP)*. TCP ensured that messages were properly routed from sender to receiver and that those messages arrived intact.

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks to facilitate both intra-organization (i.e., within the organization) and inter-organization (i.e., between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these diverse products to communicate with each other. ARPA accomplished this by developing the *Internet Protocol* (*IP),* which created a true "network of networks," the current architecture of the Internet. The combined set of protocols is now commonly called *TCP/IP*.

Initially, use of the Internet was limited to universities and research institutions; later, the military adopted the technology. Eventually, the government decided to allow access to the Internet for commercial purposes. When this decision was made, there was resentment among the research and military communities—it was felt that response times would become poor as "the Net" became saturated with so many users.

In fact, the opposite has occurred. Businesses rapidly realized that, by making effective use of the Internet, they could refine their operations and offer new and better services to their clients. Companies started spending vast amounts of money to develop and enhance their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. The result is that *bandwidth* (i.e., the information-carrying capacity of communications lines) on the Internet has increased tremendously, while hardware costs have plummeted. It is widely believed that the Internet played a significant role in the economic growth that many industrialized nations experienced over the last decade.

The *World Wide Web* allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animations, audios and/or videos) on almost any subject. Even though the Internet was developed more than three decades ago, the introduction of the World Wide Web (WWW) was a relatively recent event. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began to develop a technology for sharing information via hyperlinked text documents. Basing the new language on the well-established *Standard Generalized Markup Language* (*SGML*)—a standard for business data interchange—Berners-Lee called his invention the *HyperText*

Markup Language (*HTML*). He also wrote communication protocols to form the backbone of his new hypertext information system, which he referred to as the *World Wide Web*.

Historians will surely list the Internet and the World Wide Web among the most important and profound creations of humankind. In the past, most computer applications ran on "stand-alone" computers (computers that were not connected to one another). Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet and World Wide Web merge computing and communications technologies, expediting and simplifying our work. They make information instantly and conveniently accessible to large numbers of people. They enable individuals and small businesses to achieve worldwide exposure. They are profoundly changing the way we do business and conduct our personal lives.

1.14 World Wide Web Consortium (W3C)

In October 1994, Tim Berners-Lee founded an organization, called the *World Wide Web Consortium* (*W3C*), that is devoted to developing nonproprietary, interoperable technologies for the World Wide Web. One of the W3C's primary goals is to make the Web universally accessible—regardless of disabilities, language or culture.

The W3C is also a standardization organization and is comprised of three *hosts*—the Massachusetts Institute of Technology (MIT), France's INRIA (Institut National de Recherche en Informatique et Automatique) and Keio University of Japan—and over 400 members, including Deitel & Associates, Inc. Members provide the primary financing for the W3C and help provide the strategic direction of the Consortium. To learn more about the W3C, visit **`www.w3.org`**.

Web technologies standardized by the W3C are called *Recommendations*. Current W3C Recommendations include *Extensible HyperText Markup Language (XHTML™)*, *Cascading Style Sheets (CSS™)* and the *Extensible Markup Language (XML)*. Recommendations are not actual software products, but documents that specify the role, syntax and rules of a technology. Before becoming a W3C Recommendation, a document passes through three major phases: *Working Draft*—which, as its name implies, specifies an evolving draft; *Candidate Recommendation*—a stable version of the document that industry can begin to implement; and *Proposed Recommendation*—a Candidate Recommendation that is considered mature (i.e., has been implemented and tested over a period of time) and is ready to be considered for W3C Recommendation status. For detailed information about the W3C Recommendation track, see "6.2 The W3C Recommendation track" at

```
www.w3.org/Consortium/Process/Process-19991111/
process.html#RecsCR
```

1.15 Extensible Markup Language (XML)

As the popularity of the Web exploded, HTML's limitations became apparent. HTML's lack of *extensibility* (the ability to change or add features) frustrated developers, and its ambiguous definition allowed erroneous HTML to proliferate. In response to these problems, the W3C added limited extensibility to HTML. This was, however, only a temporary solution—the need for a standardized, fully extensible and structurally strict language was apparent. As a result, XML was developed by the W3C. XML combines the power and extensibility of its parent language, Standard Generalized Markup Language (SGML), with the simplicity that the Web community demands. At the same time, the W3C began developing XML-based standards for style sheets and advanced hyperlinking. *Extensible Stylesheet Language* (*XSL*) incorporates elements of both Cascading Style Sheets (CSS), which is used to format HTML documents and *Document Style and Semantics Specification Language* (*DSSSL*), which is used to format SGML documents. Similarly, the *Extensible Linking Language* (*XLink*) combines ideas from *HyTime* and the *Text Encoding Initiative* (*TEI*), to provide extensible linking of resources.

Data independence, the separation of content from its presentation, is the essential characteristic of XML. Because an XML document describes data, any application conceivably can process an XML document. Recognizing this, software developers are integrating XML into their applications to improve Web functionality and interoperability. XML's flexibility and power make it perfect for the middle tier of client/server systems, which must interact with a wide variety of clients. Much of the processing that was once limited to server computers now can be performed by client computers, because XML's semantic and structural information enables it to be manipulated by any application that can process text. This reduces server loads and network traffic, resulting in a faster, more efficient Web.

XML is not limited to Web applications. Increasingly, XML is being employed in databases—the structure of an XML document enables it to be integrated easily with database applications. As applications become more Web enabled, it seems likely that XML will become the universal technology for data representation. All applications employing XML would be able to communicate, provided that they could understand each other's XML markup, or *vocabulary*.

Simple Object Access Protocol (SOAP) is a technology for the distribution of objects (marked up as XML) over the Internet. Developed primarily by Microsoft and DevelopMentor, SOAP provides a framework for expressing application semantics, encoding that data and packaging it in modules. SOAP has three parts: The *envelope*, which describes the content and intended recipient of a SOAP message; the SOAP *encoding rules*, which are XML-based; and the SOAP *Remote Procedure Call* (*RPC*) *representation* for commanding other computers to perform a task. Microsoft .NET (discussed in the next two sections) uses XML and SOAP to mark up and transfer data over the Internet. XML and SOAP are at the core of .NET—they allow software components to interoperate (i.e., communicate easily with one another). SOAP is supported by many platforms, because of its foundations in XML and HTTP. We discuss XML in Chapter 18, Extensible Markup Language (XML) and SOAP in Chapter 21, ASP .NET and Web Services.

1.16 Introduction to Microsoft .NET

In June 2000, Microsoft announced its *.NET initiative*, a broad new vision for embracing the Internet and the Web in the development, engineering and use of software. One key aspect of the .NET strategy is its independence from a specific language or platform. Rather than forcing developers to use a single programming language, developers can create a .NET application in any .NET-compatible language. Programmers can contribute to the same software project, writing code in the .NET languages (such as C#, Visual C++ .NET, Visual Basic .NET and many others) in which they are most competent. Part of the initiative includes Microsoft's *Active Server Pages (ASP) .NET* technology, which allows programmers to create applications for the Web.

The .NET architecture can exist on multiple platforms, further extending the portability of .NET programs. In addition, the .NET strategy involves a new program-development process that could change the way programs are written and executed, leading to increased productivity.

A key component of the .NET architecture is *Web services*, which are applications that can be used over the Internet. Clients and other applications can use these Web services as reusable building blocks. One example of a Web service is Dollar Rent a Car's reservation system.[3] An airline partner wanted to enable customers to make rental-car reservations from the airline's Web site. To do so, the airline needed to access Dollar's reservation system. In response, Dollar created a Web service that allowed the airline to access Dollar's database and make reservations. Web services enable the two companies to communicate over the Web, even though the airline uses UNIX systems and Dollar uses Microsoft Windows. Dollar could have created a one-time solution for that particular airline, but the company would not have been able to reuse such a customized system. By creating a Web service, Dollar can allow other airlines or hotels to use its reservation system without creating a custom program for each relationship.

The .NET strategy extends the concept of software reuse to the Internet, allowing programmers to concentrate on their specialties without having to implement every component of every application. Instead, companies can buy Web services and devote their time and energy to developing their products. The .NET strategy further extends the concept of software reuse to the Internet by allowing programmers to concentrate on their specialties without having to implement every component. Visual programming (discussed in Chapter 2) has become popular, because it enables programmers to create applications easily, using such prepackaged components as buttons, text boxes and scrollbars. Similarly, programmers may create an application using Web services for databases, security, authentication, data storage and language translation without having to know the internal details of those components.

The .NET strategy incorporates the idea of software reuse. When companies link their products in this way, a new user experience emerges. For example, a single application could manage bill payments, tax refunds, loans and investments, using Web services from various companies. An online merchant could buy Web services for online credit-card payments, user authentication, network security and inventory databases to create an e-commerce Web site.

The keys to this interaction are XML and SOAP, which enable Web services to communicate. XML gives meaning to data, and SOAP is the protocol that allows Web services to communicate easily with one another. XML and SOAP act as the "glue" that combines various Web services to form applications.

Universal data access is another essential concept in the .NET strategy. If two copies of a file exist (such as on a personal and a company computer), the less recent version must constantly be updated—this is called file *synchronization*. If the separate versions of the file are different, they are *unsynchronized*, a situation that could lead to errors. Under .NET, data could reside in one central location rather than on separate systems. Any Internet-connected device could access the data (under tight control, of course), which would then be formatted appropriately for use or display on the accessing device. Thus, the same document could be seen and edited on a desktop PC, a PDA, a cell phone or other device. Users would not need to synchronize the information, because it would be fully up-to-date in a central area.

---

[3] Microsoft Corporation, "Dollar Rent A Car E-Commerce Case Study on Microsoft Business," 1 July 2001 **<www.microsoft.com/BUSINESS/casestudies/b2c/dollarrentacar.asp>**.

Microsoft's *HailStorm Web services* facilitate such data organization.[4] HailStorm allows users to store data so that it is accessible from any HailStorm-compatible device (such as a PDA, desktop computer or cell phone). HailStorm offers a suite of services, such as an address book, e-mail, document storage, calendars and a digital wallet. Third-party Web services also can interact with HailStorm—users can be notified when they win online auctions or have their calendars updated if their planes arrive late. Information can be accessed from anywhere and cannot become unsynchronized. Privacy concerns increase, though, because all of a user's data resides in one location. Microsoft has addressed this issue by giving users control over their data. Users must authorize access to their data and specify the duration of that access.

Microsoft plans to create Internet-based client applications. For example, software could be distributed over the Internet on a *subscription basis*, enabling immediate corrections, updates and communication with other applications over the Internet. HailStorm provides basic services at no charge and users can pay via subscription for more advanced features.

The .NET strategy is an immense undertaking. We discuss various aspects of .NET throughout this book. Additional information is available on Microsoft's Web site (**`www.microsoft.com/net`**).

1.17 .NET Framework and the Common Language Runtime

The Microsoft® *.NET Framework* is at the heart of the .NET strategy. This framework manages and executes applications and Web services, contains a class library (called the *Framework Class Library* or *FCL*), enforces security and provides many other programming capabilities. The details of the .NET Framework are found in the *Common Language Specification* (*CLS*), which contains information about the storage of data types, objects and so on. The CLS has been submitted for standardization to ECMA (the European Computer Manufacturers Association), making it easier to create the .NET Framework for other platforms. This is like publishing the blueprints of the framework—anyone can build it, following the specifications. Currently, the .NET Framework exists only for the Windows platform, although a version is under development for the FreeBSD operating system.[5] The FreeBSD project provides a freely available and open-source UNIX-like operating system that is based on that UC Berkeley's *Berkeley System Distribution* (BSD).

The *Common Language Runtime (CLR)* is another central part of the .NET Framework—it executes C# programs. Programs are compiled into machine-specific instructions in two steps. First, the program is compiled into *Microsoft Intermediate Language (MSIL)*, which defines instructions for the CLR. Code converted into MSIL from other languages and sources can be woven together by the CLR. Then, another compiler in the CLR compiles the MSIL into machine code (for a particular platform), creating a single application.

---

[4] Microsoft Corporation, "Building User-Centric Experiences: An Introduction to Microsoft HailStorm," 30 July 2001 <**`http://www.microsoft.com/net/hailstorm.asp`**>.

[5] Microsoft Corporation, "The Microsoft Shared Source C# and CLI Specifications," 30 July 2001 **`<http://www.microsoft.com/net/sharedsourcewp.asp>`**.

Why bother having the extra step of converting from C# to MSIL, instead of compiling directly into machine language? The key reasons are portability between operating systems, interoperability between languages and execution-management features such as memory management and security.

If the .NET Framework exists (and is installed) for a platform, that platform can run any .NET program. The ability of a program to run (without modification) across multiple platforms is known as *platform independence*. Code written once can be used on another machine without modification, saving both time and money. In addition, software can target a wider audience—previously, companies had to decide whether converting their programs to different platforms (sometimes called *porting*) was worth the cost. With .NET, porting is no longer an issue.

The .NET Framework also provides a high level of *language interoperability*. Programs written in different languages are all compiled into MSIL—the different parts can be combined to create a single, unified program. MSIL allows the .NET Framework to be *language independent*, because .NET programs are not tied to a particular programming language. Any language that can be compiled into MSIL is called a *.NET-compliant language*. Figure 1.1 lists many of the current languages that support the .NET platform.[6]

Language interoperability offers many benefits to software companies. C#, Visual Basic .NET and Visual C++ .NET developers, for example, can work side-by-side on the same project without having to learn another programming language—all their code compiles into MSIL and links together to form one program. In addition, the .NET Framework can package old and new components to work together. This allows companies to reuse the code that they have spent years developing and integrate it with the new .NET code that they write. Integration is crucial, because companies cannot migrate easily to .NET unless they can stay productive, using their existing developers and software.

Programming Languages
APL
C#
COBOL
Component Pascal
Curriculum
Eiffel
Fortran
Haskell
Java
JScript
Mercury
Oberon
Oz
Pascal
Perl
Python
RPG
Scheme
Smalltalk
Standard ML
Visual Basic .NET
Visual C++ .NET

Fig. 1.1 .NET Languages .

---

[6] Table information from Microsoft Web site, `www.microsoft.com`.

Another benefit of the .NET Framework is the CLR's execution-management features. The CLR manages memory, security and other features, relieving the programmer of these responsibilities. With languages like C++, programmers must take memory management into their own hands. This leads to problems if programmers request memory and never return it—programs could consume all available memory, which would prevent applications from running. By managing the program's memory, the .NET Framework allows programmers to concentrate on program logic.

The .NET Framework also provides programmers with a huge library of classes. This library, called the Framework Class Library (FCL), can be used by any .NET language. The FCL contains a variety of reusable components, saving programmers the trouble of creating new components. This book explains how to develop .NET software with C#. Steve Ballmer, Microsoft's CEO, stated in May 2001 that Microsoft was "betting the company" on .NET. Such a dramatic commitment surely indicates a bright future for C# and its community of developers.

1.18 Tour of the Book

In this section, we tour the chapters and appendices of *C# How to Program*. In addition to the topics presented in each chapter, several of the chapters contain an Internet and World Wide Web Resources section that lists additional sources from which readers can enhance their knowledge of C# programming.

Chapter 1—Introduction to Computers, Internet, World Wide Web and C#

The first chapter familiarizes the reader with what computers are, how they work and how they are programmed. We explain the evolution of programming languages, from their origins in machine languages to the development of high-level, object-oriented languages. We overview the history of the Internet, World Wide Web and various technologies (such as HTTP, SOAP and XML) that have led to advances in how computers are used. We then discuss the development of the C# programming language and the Microsoft .NET initiative, including Web services. We explore the impact of .NET on software development and conclude by touring the remainder of the book.

Chapter 2—Introduction to the Visual Studio® .NET IDE

Chapter 2 introduces Microsoft Visual Studio .NET, an *integrated development environment* (*IDE*) for the creation of C# programs. Visual Studio .NET enables *visual programming*, in which *controls* (such as buttons or text boxes) are "dragged" and "dropped" into place, rather than added by typing code. Visual programming has led to greatly increased productivity of software developers because it eliminates many of the tedious tasks that programmers face. For example, object properties (information such as height and color) can be modified through Visual Studio .NET windows, allowing changes to be made quickly and causing the results to appear immediately on the screen. Rather than having to guess how the GUI will appear while writing a program, programmers view the GUI exactly as it will appear when the finished program runs. Visual Studio .NET also contains advanced tools for debugging, documenting and writing code. The chapter presents features of Visual Studio .NET, including its key windows, toolbox and help features and overviews the process of compiling and running programs. We provide an example of the capabilities of Visual Studio .NET by using it to create a simple Windows application without typing a single line of code.

Chapter 3—Introduction to C# Programming

This chapter introduces readers to our LIVE-CODE™ approach. Every concept is presented in the context of a complete working C# program and is followed by one or more sample outputs depicting the program's execution. In our first example, we print a line of text and carefully discuss each line of code. We then discuss fundamental tasks, such as how a program inputs data from its users and how to write arithmetic expressions. The chapter's last example demonstrates how to print a variety of character strings in a window called a message box.

Chapter 4—Control Structures: Part 1

This chapter formally introduces the principles of structured programming, a set of techniques that will help the reader develop clear, understandable, maintainable programs throughout the text. The first part of this chapter presents program-development and problem-solving techniques. The chapter demonstrates how to transform a written specification to a program by using such techniques as *pseudocode* and *top-down*, *stepwise refinement*. We then progress through the entire process, from developing a problem statement into a working C# program. The notion of algorithms is also discussed. We build on information presented in the previous chapter to create programs that are interactive (i.e., they change their behavior to suit user-supplied inputs). The chapter then introduces the use of control structures that affect the sequence in which statements are executed. Control structures produce programs that are easily understood, debugged and maintained. We discuss the three forms of program control—sequence, selection and repetition—focusing on the **`if`**/**`then`** and **`while`** control structures. Flowcharts (i.e., graphical representations of algorithms) appear throughout the chapter, reinforcing and augmenting the explanations.

Chapter 5—Control Structures: Part 2

Chapter 5 introduces more complex control structures and the logical operators. It uses flowcharts to illustrate the flow of control through each control structure, including the **`for`**, **`do`**/**`while`** and **`switch`** structures. We explain the **`break`** and **`continue`** statements and the logical operators. Examples include calculating compound interest and printing the distribution of grades on an exam (with some simple error checking). The chapter concludes with a structured programming summary, including each of C#'s control structures. The techniques discussed in Chapters 4 and 5 constitute a large part of what has been taught traditionally under the topic of structured programming.

Chapter 6—Methods

A *method* allows the programmer to create a block of code that can be called upon from various points in a program. Groups of related methods can be separated into functional blocks (classes), using the "divide and conquer" strategy. Programs are divided into simple components that interact in straightforward ways. We discuss how to create our own methods that can take input, perform calculations and return output. We examine the .NET library's **`Math`** class, which contains methods (i.e., methods in a class) for performing complex calculations (e.g., trigonometric and logarithmic calculations). *Recursive* methods (methods that call themselves) and method overloading, which allows multiple methods to have the same name, are introduced. We demonstrate overloading by creating two **`Square`** methods that take an integer (i.e., whole number) and a floating-point number (i.e., a number with a decimal point), respectively. To conclude the chapter, we create a graphical simulation of the "craps" dice game, using the random-number generation techniques presented in the chapter.

Chapter 7—Arrays

Chapter 7 discusses arrays, our first data structures. (Chapter 24 discusses the topic of data structures in depth.) Data structures are crucial to storing, sorting, searching and manipulating large amounts of information. *Arrays* are groups of related data items that allow the programmer to access any element directly. Rather than creating 100 separate variables that are all related in some way, the programmer instead can create an array of 100 elements and access these elements by their location in the array. We discuss how to declare and allocate arrays, and we build on the techniques of the previous chapter by passing arrays to methods. In addition, we discuss how to pass a variable number of arguments to methods. Chapters 4 and 5 provide essential background for the discussion of arrays, because repetition structures are used to iterate through elements in the array. The combination of these concepts helps the reader create highly-structured and well-organized programs. We then demonstrate how to sort and search arrays. We discuss multidimensional arrays (both rectangular and jagged), which can be used to store tables of data.

Chapter 8—Object-Based Programming

Chapter 8 serves as our introduction into the powerful concepts of objects and *classes* (classes are programmer-defined types). As mentioned in Chapter 1, object technology has led to considerable improvements in software development, allowing programmers to create reusable components. In addition, objects allow programs to be organized in natural and intuitive ways. In this chapter, we present the fundamentals of object-based programming, such as encapsulation, data abstraction and abstract data types (ADTs). These techniques hide the details of components so that the programmer can concentrate on the “big picture.” To demonstrate these concepts, we create a time class, which displays the time in standard and military formats. Other topics examined include abstraction, composition, reusability and inheritance. We overview how to create reusable software components with assemblies, namespaces and Dynamic Link Library (DLL) files. You will learn how to create classes like those in the Framework Class Library. Other C# features discussed include properties and the **`readonly`** and **`const`** keywords. This chapter lays the groundwork for the next two chapters, which introduce object-oriented programming.

Chapter 9—Object-Oriented Programming: Inheritance

In this chapter, we discuss inheritance—a form of software reusability in which classes (called *derived classes*) are created by absorbing attributes and methods of existing classes (called *base classes*). The inherited class (i.e., the derived class) can contain additional attributes and methods. We show how finding the commonality between classes of objects can reduce the amount of work it takes to build large software systems. These proven techniques help programmers create and maintain software systems. A detailed case study demonstrates software reuse and good programming techniques by finding the commonality among a three-level inheritance hierarchy: the point, circle and cylinder classes. We discuss the software engineering benefits of object-oriented programming. The reader learns important object-oriented programming fundamentals, such as creating and extending customized classes.

Chapter 10—Object-Oriented Programming: Polymorphism

Chapter 10 continues our formal introduction of object-oriented programming. We discuss polymorphic programming and its advantages. *Polymorphism* permits classes to be treated in a general manner, allowing the same method call to act differently depending on context (e.g., “move” messages sent to a bird and a fish result in dramatically different types of action—a bird flies and a fish swims). In addition to treating existing classes in a general manner, polymorphism allows new classes to be added to a system easily. We identify situations in which polymorphism is useful. A payroll system case study demonstrates polymorphism—the system determines the wages for each employee differently to suit the type of employee (bosses paid fixed salaries, hourly workers paid by the hour, commission workers who receive a base salary plus commission and piece workers who are paid per item produced). These programming techniques and those of the previous chapter allow the programmer to create extensible and reusable software components.

Chapter 11—Exception Handling

Exception handling is one of the most important topics in C# from the standpoint of building mission-critical and business-critical applications. People can enter incorrect data, data can be corrupted and clients can try to access records that do not exist or are restricted. A simple division-by-zero error may cause a calculator program to crash, but what if such an error occurs in the navigation system of a flying airplane? Programmers must deal with these situations, because in some cases, the results of program failure could be disastrous. Programmers need to know how to recognize the errors (*exceptions*) that could occur in software components and handle those exceptions effectively, allowing programs to deal with problems and continue executing instead of “crashing.” This chapter overviews the proper use of exception handling and various exception-handling techniques. We cover the details of C# exception handling, the termination model of exception handling, throwing and catching exceptions, and library class **`Exception`**. Programmers who construct software systems from reusable components built by other programmers often deal with the exceptions that those components may throw.

Chapter 12—Graphical User Interface Concepts: Part 1

Chapter 12 explains how to add graphical user interfaces (GUIs) to our programs, providing a professional look and feel. By using the techniques of rapid application development (RAD), we can create a GUI from reusable components, rather than explicitly programming every detail. The Visual Studio .NET IDE makes developing GUIs even easier by allowing the programmer to position components in a window through so-called visual programming. We discuss how to construct user interfaces with *Windows Forms* GUI components such as labels, buttons, text boxes, scroll bars and picture boxes. We also introduce *events*, which are messages sent by a program to signal to an object or a set of objects that an action has occurred. Events are most commonly used to signal user interactions with GUI components, but also can signal internal actions in a program. We overview event handling and discuss how to handle events specific to controls, the keyboard and the mouse. Tips are included throughout the chapter to help the programmer create visually appealing, well-organized and consistent GUIs.

Chapter 13—Graphical User Interface Concepts: Part 2

Chapter 13 introduces more complex GUI components, including menus, link labels, panels, list boxes, combo boxes and tab controls. In a challenging exercise, readers create an application that displays a disk drive’s directory structure in a tree—similar to that created by Windows Explorer. The *Multiple Document Interface* (*MDI*) is presented, which allows multiple documents (i.e., forms) to be open simultaneously in a single GUI. We conclude with a discussion of how to create custom controls by combining existing controls. The techniques presented in this chapter allow readers to create sophisticated and well-organized GUIs, adding style and usability to their applications.

Chapter 14—Multithreading
We have come to expect much from our applications. We want to download files from the Internet, listen to music, print documents and browse the Web—all at the same time! To do this, we need a technique called *multithreading*, which allows applications to perform multiple activities concurrently. C# includes built-in capabilities to enable multithreaded applications, while shielding programmers from complex details. C# is better equipped to deal with more sophisticated multimedia, network-based and multiprocessor-based applications than other languages that do not have multithreading features. This chapter overviews the built-in threading classes of C# and covers threads, thread life-cycles, time-slicing, scheduling and priorities. We analyze the producer-consumer relationship, thread synchronization and circular buffers. This chapter lays the foundation for creating the impressive multithreaded programs that clients demand.

Chapter 15—Strings, Characters and Regular Expressions
In this chapter, we discuss the processing of words, sentences, characters and groups of characters. In C#, ***`string`***s (groups of characters) are objects. This is yet another benefit of C#'s emphasis on object-oriented programming. Objects of type **`string`** contain methods that can copy, create hash codes, search, extract substrings and concatenate strings with one another. As an interesting example of strings, we create a card shuffling-and-dealing simulation. We discuss regular expressions, a powerful tool for searching and manipulating text.

Chapter 16—Graphics and Multimedia
In this chapter, we discuss *GDI+* (an extension of the *Graphics Device Interface—GDI*), the Windows service that provides the graphical features used by .NET. The extensive graphical capabilities of GDI+ can make programs more visual and fun to create and use. We discuss C#'s treatment of graphics objects and color control, and we discuss how to draw arcs, polygons and other shapes. We use various pens and brushes to create color effects and include an example demonstrating gradient fills and textures. This chapter introduces techniques for turning text-only applications into exciting, aesthetically pleasing programs that even novice programmers can write with ease. The second half of the chapter focuses on audio, video and speech technology. We discuss adding sound, video and animated characters to programs (primarily using existing audio and video clips). You will see how easy it is to incorporate multimedia into C# applications. This chapter introduces an exciting technology called *Microsoft Agent* for adding *interactive animated characters* to a program. Each character allows users to interact with the application, using more natural human communication techniques, such as speech. The agent characters accept mouse and keyboard interaction, speak and hear (i.e., they support speech synthesis and speech recognition). With these capabilities, your applications can speak to users and actually respond to their voice commands!

Chapter 17—Files and Streams
Imagine a program that could not save data to a file. Once the program is closed, all the work performed in the program is lost forever. For this reason, this chapter is one of the

most important for programmers who will be developing commercial applications. We explain how to input and output streams of data from and to files, respectively. We discuss how programs read and write data from and to secondary storage devices (such as disks). A detailed example demonstrates these concepts by allowing the user to read and write bank account information to and from files. We introduce those classes and methods in C# that help perform input and output conveniently—they demonstrate the power of object-oriented programming and reusable classes. We discuss benefits of sequential files, random-access files and buffering. This chapter is crucial for developing C# file-processing applications and networking applications (Chapter 22), which also use the techniques in this chapter to send and receive data.

Chapter 18—Extensible Markup Language (XML)[7]

The Extensible Markup Language (XML) derives from SGML (Standardized General Markup Language), which became an industry standard in 1986. Although SGML is employed in publishing applications worldwide, it has not been incorporated into the mainstream computing and information technology curricula because of its sheer size and complexity. XML is an effort to make SGML-like technology available to a much broader community. It was created by the World Wide Web Consortium (W3C) for describing data in a portable format, is one of most important technologies in industry today and is being integrated into almost every field. XML differs in concept from markup languages such as the HyperText Markup Language (HTML). HTML is a markup language for describing how information is rendered in a browser. XML is a language for creating markup languages for virtually any type of information. Document authors use XML to create entirely new markup languages to describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes. Markup languages created with XML include WML (Wireless Markup Language), XHTML (Extensible HyperText Markup Language, for Web content), MathML (for mathematics), VoiceXML™ (for speech), SMIL™ (Synchronized Multimedia Integration Language, for multimedia presentations), CML (Chemical Markup Language, for chemistry) and XBRL (Extensible Business Reporting Language, for financial data exchange). Companies and individuals constantly are finding new and exciting uses for XML. In this chapter, we present examples that illustrate the basics of marking up data with XML. We demonstrate several XML-derived markup languages, such as *XML Schema* (for checking an XML document's grammar), *XSLT (Extensible Stylesheet Language Transformations*, for transforming an XML document's data into another text-based format such as XHTML) and Microsoft's *BizTalk*™ (for marking up business transactions). (For readers who are unfamiliar with XHTML, we provide Appendices K and L, which provide a detailed introduction to XHTML.)

Chapter 19—Database, SQL and ADO .NET

Access and storage of data are integral to creating powerful software applications. This chapter discusses .NET support for database manipulation. Today's most popular database systems are relational databases. In this chapter, we introduce the Structured Query Language (SQL) for performing queries on relational databases. We introduce ADO .NET—an extension of Microsoft's ActiveX Data Objects that enables .NET applications to access and manipulate databases. ADO .NET allows data to be exported as XML, which enables applications that use ADO .NET to communicate with a variety of programs that understand XML. The reader will learn how to create database connections, using tools provided in Visual Studio .NET, and will learn how to use the classes in the **`System.Data`** namespace.

---

[7] The reader interested in a deeper treatment of XML may want to consider our book, *XML How to Program*.

Chapter 20—ASP .NET, Web Forms and Web Controls

Previous chapters demonstrated how to create applications that execute locally on the user's computer. In this chapter and the next, we discuss how to create Web-based applications using *Active Server Pages (ASP) .NET*. This is a crucial aspect of .NET and of Microsoft's vision of how software should be deployed on the Internet. ASP .NET is an integral technology for creating dynamic Web content marked up as HTML. (For readers who are unfamiliar with HTML, we provide a detailed introduction in Appendices I and J.) *Web Forms* provide GUIs for ASP .NET pages and can contain *Web controls*, such as labels, buttons and text boxes with which users interact. Like Windows Forms, Web Forms are designed using visual programming. This chapter presents many interesting examples, which include an online guest book application and a multi-tier, database intensive application that allows users to query a database for a list of publications by a specific author. Debugging Web Forms using the **`Trace`** property is also discussed.

Chapter 21—ASP .NET and Web Services

Chapter 21 continues our discussion of ASP .NET. In this chapter, we introduce *Web services*, which are programs that "expose" services (i.e., methods) to clients. Using Web services, programmers can create methods that anyone can invoke. This enables applications to invoke methods remotely over a network. Web services offer increased software reusability, making the Internet, in essence, a programming library available to programmers worldwide. Web services use XML and SOAP to mark up and send information, respectively. This chapter presents several interesting examples that include Web services for manipulating huge numbers (up to 100 digits), simulating the card game of blackjack and implementing an airline reservation system. One particularly interesting example is our temperature server, a Web service that gathers weather information for dozens of cities in the United States.

Chapter 22—Networking: Streams-Based Sockets and Datagrams

Chapter 22 introduces the fundamental techniques of C#-based networking—streams and datagrams. We demonstrate how streams-based *sockets* allow us to hide many networking details. With sockets, networking is as simple as if we were reading from and writing to a file. We also introduce *datagrams* in which packets of information are sent between programs. Each packet is addressed to its recipient and sent out to the network, which routes the packet to its destination. The examples in this chapter focus on communication between applications. One example demonstrates using streams-based sockets to communicate between two C# programs. Another similar example, sends datagrams between applications. We also show how to create a multithreaded-server application that can communicate multiple clients in parallel. In this client/server tic-tac-toe game, the server maintains the status of the game and two clients communicate with the server to play the game.

Chapter 23—Data Structures and Collections

This chapter discusses arranging data into aggregations such as linked lists, stacks, queues and trees. Each data structure has properties that are useful in a wide variety of applications, from sorting elements to keeping track of method calls. We discuss how to build each of these data structures. This is also a valuable experience in crafting useful classes. In addition, we cover prebuilt collection classes in the .NET Framework Class Library. These data structures have many useful methods for sorting, inserting, and deleting items, plus methods to enable data structures to resize themselves dynamically. When possible, C# programmers should use the Framework Class Library to find appropriate data structures, rather than implementing these data structures themselves. This chapter reinforces much of the object technology discussed in Chapters 8, 9 and 10, including classes, inheritance and composition.

Chapter 24—Accessibility

The World Wide Web presents a challenge to individuals with disabilities. Multimedia-rich Web sites are difficult for text readers and other programs to interpret; thus, users with hearing and visual impairments may have difficulty browsing such sites. To help rectify this situation, the World Wide Web Consortium (W3C) launched the *Web Accessibility Initiative (WAI)*, which provides guidelines for making Web sites accessible to people with disabilities. This chapter provides a description of these guidelines, such as the use of the **`<headers>`** tag to make tables more accessible to page readers, the **`alt`** attribute of the **`<img>`** tag to describe images, and XHTML and CSS to ensure that a page can be viewed on almost any type of display or reader. We illustrate key accessibility features of Visual Studio .NET and of Windows 2000. We also introduce *VoiceXML* and *CallXML*, two technologies for increasing the accessibility of Web-based content. VoiceXML helps people with visual impairments to access Web content via speech synthesis and speech recognition. CallXML allows users with visual impairments to access Web-based content through a telephone. In the chapter exercises, readers create their own voice mail applications, using CallXML.

Appendix A—Operator precedence chart

This appendix lists C# operators and their precedence.

Appendix B—Number Systems

This appendix explains the binary, octal, decimal and hexadecimal number systems. It also reviews the conversion of numbers among these bases and illustrates mathematical operations in each base.

Appendix C—Career Opportunities

This appendix provides career resources for C# programmers.

Appendix D—Visual Studio .NET Debugger

This appendix introduces the Visual Studio .NET debugger for locating logic errors in programs. Key features of this appendix include setting breakpoints, stepping through programs line-by-line and “watching” variable values.

Appendix E—Generating Documentation in Visual Studio

Appendix E discusses how to create comments within C# code that can be extracted to create powerful, XML-based documentation.

Appendix F—ASCII Character Set

This appendix contains a table of the 128 ASCII alphanumeric symbols and their corresponding ASCII (American Standard Code for Information Interchange) numbers.

Appendix G—Unicode®

This appendix introduces the Unicode Standard, an encoding scheme that assigns unique numeric values to the characters of most of the world's languages. We include a Windows application that uses Unicode encoding to print welcome messages in several different languages.

Appendix H—COM Integration

Prior to .NET, COM (Component Object Model) was critical for specifying how different Windows programming languages communicate at the binary level. For example, COM components such as ActiveX controls and ActiveX DLLs often were written in Microsoft Visual C++, but used in other programs. The .NET platform does not directly support COM components, but Microsoft provides tools for the integration of COM components with .NET applications. In this appendix, we explore some of these tools by integrating an ActiveX control and an ActiveX DLL into C# applications.

Appendices I and J—Introduction to HyperText Markup Language 4: 1 & 2 (on CD)

These appendices provide an introduction to *HTML*—the *Hypertext Markup Language*. HTML is a *markup language* for describing the elements of an HTML document (Web page) so that a browser, such as Microsoft's Internet Explorer, can render (i.e., display) that page. These appendices are included for our readers who do not know HTML or who would like a review of HTML before studying Chapter 20, ASP .NET, Web Forms and Web Controls. We do not present any C# programming in these appendices. Some key topics covered in Appendix I include: incorporating text and images in an HTML document, linking to other HTML documents on the Web, incorporating special characters (such as copyright and trademark symbols) into an HTML document and separating parts of an HTML document with horizontal lines (called *horizontal rules*). In Appendix J, we discuss more substantial HTML elements and features. We demonstrate how to present information in *lists* and *tables*. We discuss how to collect information from people browsing a site. We explain how to use *internal linking* and *image maps* to make Web pages easier to navigate. We also discuss how to use *frames* to display multiple documents in the browser window.

Appendices K and L—Introduction to XHTML: Parts 1 & 2 (on CD)

In these appendices, we introduce the Extensible Hypertext Markup Language (XHTML). XHTML is an emerging W3C technology designed to replace HTML as the primary means of describing Web content. As an XML-based language, XHTML is more robust and extensible than HTML. XHTML incorporates most of HTML 4's elements and attributes—the focus of these appendices. Appendices K and L are included for our readers who do not know XHTML or who would like a review of XHTML before studying Chapter 18, Extensible Markup Language (XML) and Chapter 24, Accessibility.

Appendix M—HTML/XHTML Special Characters (on CD)

This appendix provides many commonly used HTML/XHTML special characters, called *character entity references*.

Appendix N—HTML/XHTML Colors (on CD)

This appendix lists commonly used HTML/XHTML color names and their corresponding hexadecimal values.

Appendix O—Bit Manipulation

This appendix discusses C#'s powerful bit-manipulation capabilities. This helps programs process bit strings, set individual bits on or off and store information more compactly. Such capabilities—inherited from C—are characteristic of low-level assembly languages and are valued by programmers writing systems software, such as operating system and networking software.

1.19 Internet and World Wide Web Resources

`www.deitel.com`

This is the official Deitel & Associates, Inc. Web site. Here you will find updates, corrections, downloads and additional resources for all Deitel publications. In addition, this site provides information about Deitel & Associates, Inc. professional, on-site seminars offered worldwide. In the near future, you will be able to register here to receive the *Deitel Buzz* e-mail newsletter.

`www.prenhall.com/deitel`

This is Prentice Hall's Web site for Deitel publications, which contains information about our products and publications, downloads, Deitel curriculum and author information.

`www.InformIT.com/deitel`

This is the Deitel & Associates, Inc. page on the InformIT Web site. InformIT is an all-around resource for IT professionals providing articles, electronic publications and other resources for today's hottest technologies. The Deitel kiosk at InformIT.com will have free articles and for-purchase electronic publications. In addition, you can purchase all Deitel products at this site.

`www.w3.org`

The World Wide Web Consortium (W3C) is an organization that develops and recommends technologies for the Internet and World Wide Web. This site includes links to W3C technologies, news, mission statements and frequently asked questions (FAQs).

`www.elsop.com/wrc/h_comput.htm`

This site contains presents the history of computing, content about famous innovators, the evolution of languages and the development of operating systems.

`www.w3.org/History.html`

This site overviews the history of the Internet. After briefly covering developments from 1945 –1988, the site details technological advances on a year-by-year basis, from 1989 to the present day.

`www.netvalley.com/intval.html`

This site presents the history of the Internet and the World Wide Web.

`www.microsoft.com`

The Microsoft Corporation Web site provides information and technical resources for all Microsoft products, including .NET, enterprise software and the Windows operating system.

SUMMARY

[This chapter is primarily a summary of the rest of the book, so we have not provided a summary section. The remaining chapters include detailed summaries of their contents.]

TERMINOLOGY

action
"administrative" section of the computer
Advanced Research Projects Agency (ARPA)
algorithm
Apple Computer
arithmetic and logic unit (ALU)

assembler
assembly language
bandwidth
batch
batch processing
building-block approach
C programming language
C# programming language
C++ programming language
calculation
Cascading Style Sheets (CSS)
central processing unit (CPU)
clarity
class
class libraries
Common Language Runtime (CLR)
Common Language Specification (CLS)
compiler
component
computation
computer
computer program
computer programmer
data
data independence
decision
disk
distributed computing
ECMA (European Computer
    Manufacturer’s Association)
e-mail (electronic mail)
Framework Class Library (FCL)
functionalization
HailStorm Web service
hardware
hardware platform
high-level language
HTML (HyperText Markup Language)
HTTP (HyperText Transfer Protocol)
IBM (International Business Machines)
input device
input unit
Internet
interpreter
intranet
IP (Internet Protocol)
Java programming language
job
keyboard
language independence
language interoperability
live-code™ approach
logical decision
logical unit
machine dependent
machine language
maintenance of software
“manufacturing” section of the computer
memory
memory unit
Microsoft .NET
Microsoft Intermediate Language (MSIL)
mouse
multiprogramming
multitasking
.NET Framework
.NET initiative
.NET language
n-tier application
object
object-based programming
object-oriented language
object-oriented programming (OOP)
operating system
output device
output unit
Pascal programming language
personal computer
platform independence
portability
porting
primary memory
processing unit
program
programmer
property of an object
“receiving” section of the computer
reusable software component
screen
secondary storage
share the resources of a computer
“shipping” section of the computer
silicon chip
SOAP (Simple Object Access Protocol)
software
software component
software reuse
structured programming
subscription-based software
task

TCP (Transmission Control Protocol)
TCP/IP (Transmission Control Protocol/Internet Protocol)
terminal
throughput
timesharing
translator program
universal data access
UNIX
virtual-memory operating system
Visual Basic .NET programming language
visual programming
W3C (World Wide Web Consortium)
W3C Recommendation
"warehouse" section of the computer
Web Form
Web service
Web site
Win32 API (Windows 32-bit Application Programming Interface)
World Wide Web (WWW)
XML (Extensible Markup Language)

SELF-REVIEW EXERCISES

1.1 Fill in the blanks in each of the following statements:

a) A computer can directly understand only its native __________ language, which is composed only of 1s and 0s.
b) Computers process data under the control of sets of instructions called computer __________.
c) SOAP is an acronym for __________.
d) __________ is a technology derived from SGML that is used to create mark up languages.
e) The three types of languages discussed in the chapter are machine languages, __________ and __________.
f) Programs that translate high-level language programs into machine language are called __________.
g) Visual Studio .NET is a/an __________ (IDE) in which C# programs are developed.
h) C is widely known as the development language of the __________ operating system.
i) The __________ provides a large programming library for .NET languages.
j) The Department of Defense developed the Ada language with a capability called multitasking, which allows programmers to specify activities that can proceed in parallel. C# offers a similar capability called __________.
k) Web services use __________ and __________ to mark up and send information over the Internet, respectively.

1.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Universal data access is an essential part of .NET.
b) W3C standards are called recommendations.
c) C# is an object-oriented language.
d) The Common Language Runtime (CLR) requires that programmers manage their own memory.
e) C# is the only language available for programming .NET applications.
f) Procedural programming models the world more naturally than object-oriented programming.
g) Computers can directly understand high-level languages.
h) MSIL is the common intermediate format to which all .NET programs compile, regardless of their original .NET language.
i) The .NET Framework is portable to non-Windows platforms.
j) Compiled programs run faster than their corresponding interpreted programs.
k) Throughput is the amount of work a computer can process in a given time period.

ANSWERS TO SELF-REVIEW EXERCISES

1.1 a) machine. b) programs. c) Simple Object Access Protocol. d) XML. e) assembly languages, high-level languages. f) compilers. g) integrated development environment (IDE). h) UNIX. i) Framework Class Library (FCL). j) multithreading. k) XML, SOAP.

1.2 a) True. b) True. c) True. d) False. The CLR handles memory management. e) False. C# is one of many .NET languages (others include Visual Basic and Visual C++). f) False. Object-oriented programming is a more natural way to model the world than is procedural programming. g) False. Computers can directly understand only their own machine languages. h) True. i) True. j) True. k) True.

EXERCISES

1.3 Categorize each of the following items as either hardware or software:

a) CPU.
b) Compiler.
c) Input unit.
d) A word-processor program.
e) A Visual Basic .NET program.

1.4 Distinguish between the terms HTML, XML and XHTML.

1.5 Translator programs, such as assemblers and compilers, convert programs from one language (referred to as the source language) to another language (referred to as the object language or target language). Determine which of the following statements are *true* and which are *false*:

a) An assembler translates source language programs into machine language programs.
b) A compiler converts source-language programs into object-language programs.
c) High-level languages are generally machine dependent.
d) A machine-language program requires translation before it can be run on a computer.
e) The Visual Basic .NET compiler translates a high-level language into SMIL.

1.6 What are the basic requirements of a .NET language? What is needed to run a .NET program on a new type of computer (machine)?

1.7 Expand each of the following acronyms:

a) W3C.
b) XML.
c) SOAP.
d) TCP/IP.
e) OOP.
f) CLR.
g) CLS.
h) FCL.
i) MSIL.

1.8 What are the key benefits of the .NET Framework and the CLR? What are the drawbacks?

2

Introduction to the Visual Studio .NET IDE

Objectives

- To become familiar with the Visual Studio .NET Integrated development environment (IDE).
- To become familiar with the types of commands contained in the IDE's menus and toolbars.
- To identify and understand the use of various kinds of windows in Visual Studio .NET.
- To understand the features provided by the toolbar.
- To understand Visual Studio .NET's help features.
- To be able to create, compile and execute a simple C# program.

Seeing is believing.
Proverb

Form ever follows function.
Louis Henri Sullivan

Intelligence… is the faculty of making artificial objects, especially tools to make tools.
Henri-Louis Bergson

Outline

2.1 Introduction
2.2 Visual Studio .NET Integrated Development Environment (IDE) Overview
2.3 Menu Bar and Toolbar
2.4 Visual Studio .NET Windows
  2.4.1 Solution Explorer
  2.4.2 Toolbox
  2.4.3 Properties Window
2.5 Using Help
2.6 Simple Program: Displaying Text and an Image

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

2.1 Introduction

Visual Studio .NET is Microsoft's integrated development environment (IDE) for creating, documenting, running and debugging programs written in a variety of .NET programming languages. Visual Studio .NET also offers editing tools for manipulating several types of files. Visual Studio .NET is a powerful and sophisticated tool for creating business-critical and mission-critical applications. In this chapter, we provide an overview of the Visual Studio .NET features needed to create a simple C# program. We introduce additional IDE features throughout the book.

2.2 Visual Studio .NET Integrated Development Environment (IDE) Overview

When Visual Studio .NET is executed for the first time, the ***Start Page*** is displayed (Fig. 2.1). This page contains helpful links, which appear on the left side of the **Start Page**. Users can click the name of a section (such as **Get Started**) to browse its contents. We refer to single-clicking with the left mouse button as *selecting* or *clicking* and to clicking twice with the left mouse button as *double-clicking*. [*Note*: The user should be aware that there are slight differences in the way Visual Studio appears based on the version being used.]

The ***Get Started*** section contains links to recently opened projects. The most recently opened projects appear on this list (such as **ASimpleProgram** in Fig. 2.1), along with their modification dates. Alternately, the user can go to the select **Recent Projects** from the **File** menu. The first time Visual Studio .NET is loaded, this section will be empty. There are two *buttons* on the page: ***Open Project*** and ***New Project***. A button is a raised, rectangular area that performs an action when clicked.

The ***What's New*** section displays new features and updates for Visual Studio .NET, including downloads for code samples and new programming tools. The ***Online Community*** section includes ways to contact other software developers, using newsgroups, Web pages and other online resources. The ***Headlines*** section provides a way to browse news,

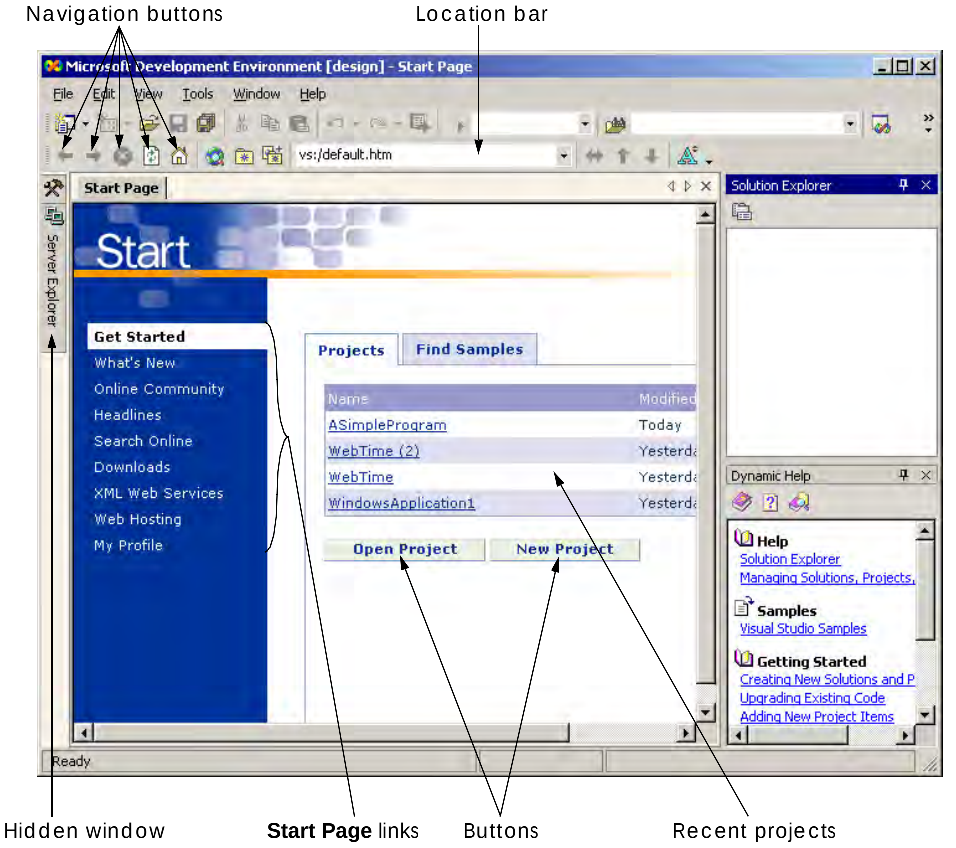

Navigation buttons
Location bar
Microsoft Development Environment [design] - Start Page
File Edit View Tools Window Help
vs:/default.htm
Start Page
Solution Explorer
Server Explorer
Start
Get Started
What's New
Online Community
Headlines
Search Online
Downloads
XML Web Services
Web Hosting
My Profile
Projects
Find Samples
Name
Modified
ASimpleProgram
Today
WebTime (2)
Yesterd
WebTime
Yesterd
WindowsApplication1
Yesterd
Open Project
New Project
Dynamic Help
Help
Solution Explorer
Managing Solutions, Projects,
Samples
Visual Studio Samples
Getting Started
Creating New Solutions and P
Upgrading Existing Code
Adding New Project Items
Ready
Hidden window
Start Page links
Buttons
Recent projects

Fig. 2.1 **Start Page** in Visual Studio .NET.

articles and how-to guides. Use the ***Search Online*** section to browse through the *MSDN* (*Microsoft Developer Network*) online library. The MSDN site includes numerous articles, downloads and tutorials for a variety of technologies. The ***Downloads*** section allows the user to obtain updates and code samples. The ***XML Web Services*** page provides programmers with information about *Web services*, which are reusable pieces of software available on the Internet. We discuss this technology in Chapter 21, ASP .NET and Web Services. ***Web Hosting*** provides information for developers who wish to post their software (such as Web services) online for public use. The ***My Profile*** page allows users to customize Visual Studio .NET, such as setting keyboard and window layout preferences. Users also can customize Visual Studio .NET selecting **Options…** or **Customize…** from the **Tools** menu. [*Note*: From this point forward, we use the **>** character to indicate the selection of a menu command. For example, we use the notation **Tools > Options…** and **Tools > Customize…** to indicate the selection of the **Options…** and **Customize…** commands, respectively.] Visual Studio .NET can even browse the Web—Internet Explorer is part of the IDE. To access a Web page, type its address into the location bar (see Fig. 2.1) and press the *Enter* key. [*Note*: The computer must be connected to the Internet.]

Several other windows appear in the IDE in addition to the **Start Page**. We discuss these windows in the following sections.

To create a new C# program, click the **New Project** button in the **Get Started** section. This action displays the *dialog* in Fig. 2.2. Dialogs are windows used to communicate with users. They typically contain buttons that allow the users to make decisions.

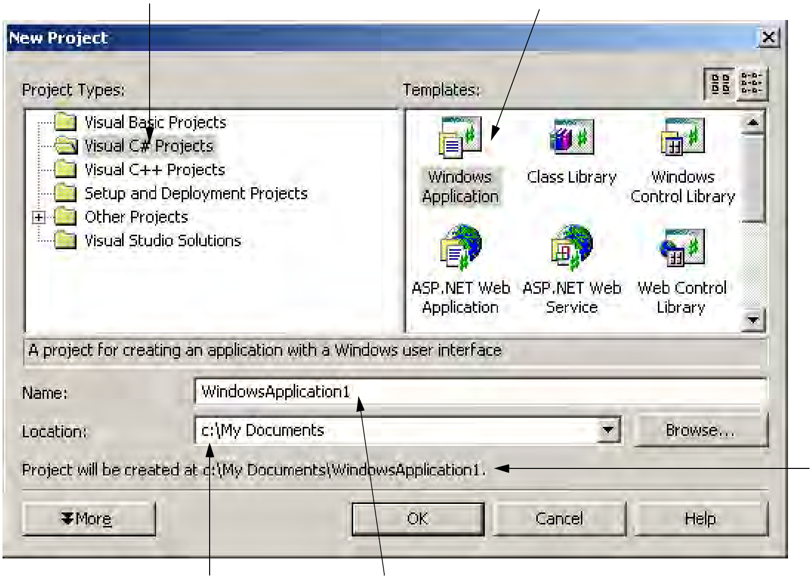
Visual Studio .NET organizes programs into *projects* and *solutions*. A project is a group of related files, such as C# code, images and documentation. A solution is a group of projects that represent a complete application, or a set of related applications. Each project in the solution may perform a different task. In this chapter, we create a single-project solution.

Visual Studio .NET allows us to create projects in a variety of programming languages. This book focuses on C#, so select the **Visual C# Projects** folder (Fig. 2.2). There are a variety of project types from which to choose, several of which are used throughout this book. In this case, create a *Windows application*. Windows applications are programs that execute inside the Windows OS, like Microsoft Word, Internet Explorer and Visual Studio .NET. Typically, they contain *controls*—graphical elements, such as buttons and labels—with which the user interacts.

By default, Visual Studio .NET assigns the name `WindowsApplication1` to the project and to the solution (Fig. 2.2). The default location for storing related files is the folder where the last project was created. The first time Visual Studio .NET executes, the default folder is the **Visual Studio Projects** folder in the **My Documents** folder. The user can change both the name and the location of the folder in which to save the project. After selecting a name and location for the project, click **OK** in the **New Project** dialog. The IDE will then change its appearance, as shown in Fig. 2.3.

Visual C# Projects folder

Visual C# **Windows application** (selected)


New Project
Project Types:
Templates:
Visual Basic Projects
Visual C# Projects
Visual C++ Projects
Setup and Deployment Projects
Other Projects
Visual Studio Solutions
Windows Application
Class Library
Windows Control Library
ASP.NET Web Application
ASP.NET Web Service
Web Control Library
Browse...
More
OK
Cancel
Help

A project for creating an application with a Windows user interface

Name: WindowsApplication1

Location: c:\My Documents

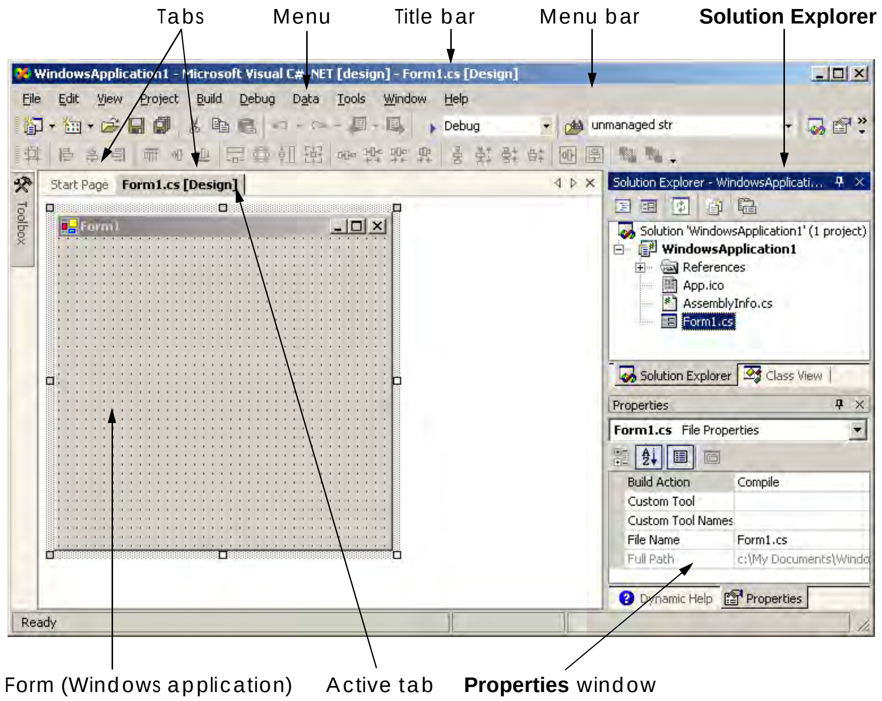
Project will be created at c:\My Documents\WindowsApplication1.

Description of selected project

Project location

Project name

Fig. 2.2 **New Project** dialog.


Tabs
Menu
Title bar
Menu bar
Solution Explorer
WindowsApplication1 - Microsoft Visual C# .NET [design] - Form1.cs [Design]
File
Edit
View
Project
Build
Debug
Data
Tools
Window
Help
Debug
unmanaged str
Toolbox
Start Page
Form1.cs [Design]
Form1
Solution Explorer - WindowsApplicati...
Solution 'WindowsApplication1' (1 project)
WindowsApplication1
References
App.ico
AssemblyInfo.cs
Form1.cs
Solution Explorer
Class View
Properties
Form1.cs File Properties
Dynamic Help
Properties
Ready
Form (Windows application)
Active tab
Properties window

Build Action	Compile
Custom Tool	
Custom Tool Names	
File Name	Form1.cs
Full Path	c:\My Documents\Windo

Fig. 2.3 Visual Studio .NET environment after a new project has been created.

The gray rectangle represents the window for our application. This rectangle is called the *form*. We discuss how to add controls to the form later in this chapter. The form and controls are the *graphical user interface* (*GUI*) of the program. They are the graphical components through which the user interacts with the program. Users enter data (*inputs*) into the program by entering information from the keyboard and by clicking the mouse buttons. The program displays instructions and other information (*outputs*) for users to read in the GUI.

The top of the IDE window (the *title bar* in Fig. 2.3) displays **WindowsApplication1 - Microsoft Visual C# .NET [design] - Form1.cs [Design]**. This title provides the name of the project (**WindowsApplication1**), the programming language (**Microsoft Visual C# .NET**), the *mode of the IDE* (**design** mode), the file being viewed (**Form1.cs**) and the mode of the file being viewed (**Design** mode). The file name `Form1.cs` is the default for Windows applications. We discuss the various modes in Section 2.6.

Notice how a tab appears for each open document. In our case, the documents are the **Start Page** and **Form1.cs [Design]**. To view a tabbed document, click the tab with the name of the document you wish to view. Tabbing saves space and allows easy access to multiple documents.

2.3 Menu Bar and Toolbar

Commands for managing the IDE and for developing, maintaining and executing programs are contained in the menus. Figure 2.4 shows the menus displayed on the menu bar. Menus contain groups of related commands that, when selected, cause the IDE to perform various actions (e.g., open a window). For example, new projects can be created by selecting **File > New > Project...** from the menu bar. The menus shown in Fig. 2.4 are summarized in Fig. 2.5. Visual Studio .NET provides different modes for the user. One of these modes is the design mode, which will be discussed later. Certain menu items appear only in specific IDE modes.

Rather than having to navigate the menus for certain commonly used commands, the programmer can access the commands from the *toolbar* (Fig. 2.6). The toolbar contains pictures called *icons* that represent commands. To execute a command, click its icon. Click the *down arrow* beside an icon to display other available options. Figure 2.6 shows the standard (default) toolbar and an icon that uses the down arrow.

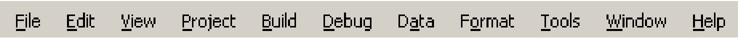

File
Edit
View
Project
Build
Debug
Data
Format
Tools
Window
Help

Fig. 2.4 Visual Studio .NET menu bar.

Menu	Description
File	Contains commands for opening projects, closing projects, printing projects, etc.
Edit	Contains commands such as cut, paste, find, undo, etc.
View	Contains commands for displaying IDE windows and toolbars.
Project	Contains commands for adding features, such as forms, to the project.
Build	Contains commands for compiling a program.
Debug	Contains commands for debugging and executing a program.
Data	Contains commands for interacting with databases.
Format	Contains commands for arranging a form's controls.
Tools	Contains commands for additional IDE tools and options for customizing the environment.
Windows	Contains commands for arranging and displaying windows.
Help	Contains commands for getting help.

Fig. 2.5 Visual Studio .NET menu summary.

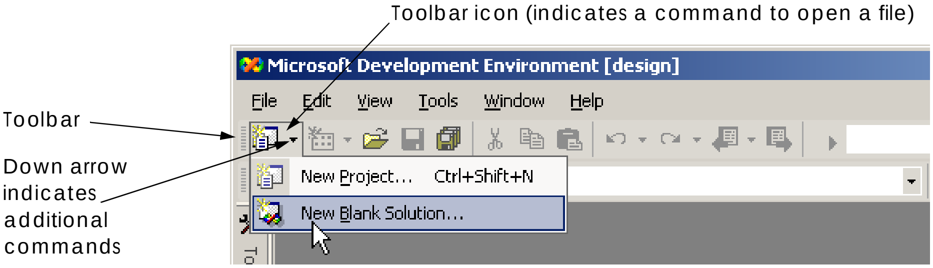

Toolbar icon (indicates a command to open a file)
Microsoft Development Environment [design]
File
Edit
View
Tools
Window
Help
Toolbar
New Project... Ctrl+Shift+N
Down arrow indicates additional commands
New Blank Solution...

Fig. 2.6 Visual Studio .NET toolbar.

Holding the mouse pointer over an icon on the toolbar highlights that icon and displays a description called a *tool tip* (Fig. 2.7). Tool tips help users understand the purposes of unfamiliar icons.

2.4 Visual Studio .NET Windows

Visual Studio .NET provides users with windows for exploring files and customizing controls. In this section, we discuss the windows that are essential for developing C# applications. These windows can be accessed using the toolbar icons below the menu bar and on the right edge of the toolbar (Fig. 2.8), or by selecting the name of the desired window from the **View** menu.

2.4.1 Solution Explorer

The **Solution Explorer** window (Fig. 2.9) lists all the files in the solution. When Visual Studio .NET is first loaded, the **Solution Explorer** is empty—there are no files to display. After a new project has been created or an existing project has been loaded, the **Solution Explorer** displays that project's contents.

The *startup project* of the solution is the project that runs when the solution is executed. It appears in bold text in the **Solution Explorer**. For our single-project solution, the startup project (**`WindowsApplication1`**) is the only project. The C# file is **`Form1.cs`**; it contains the program's code. We discuss the other files and folders later in the book.

The plus and minus boxes to the left of the project and solution names expand and collapse the tree, respectively (similar to those in Windows Explorer). Click a plus box to display more options; click a minus box to collapse a tree that already is expanded. Users also can expand or collapse a tree by double-clicking the name of the folder. Many other Visual Studio .NET windows use the plus/minus convention as well.

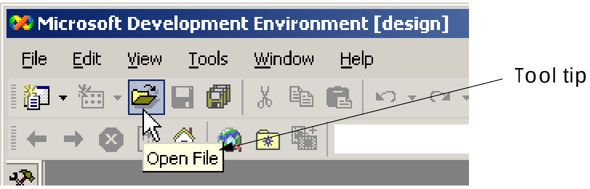

Microsoft Development Environment [design]
File Edit View Tools Window Help
Open File
Tool tip

Fig. 2.7 Tool tip demonstration.

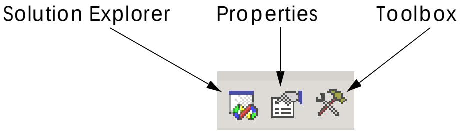
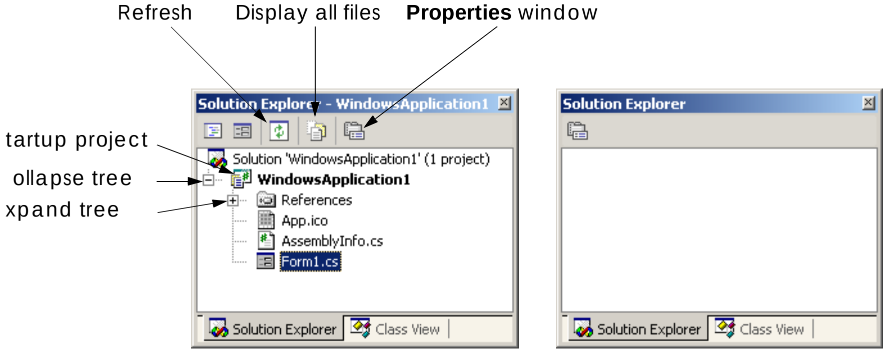


Fig. 2.8 Toolbar icons for various Visual Studio .NET windows.


Refresh
Display all files
Properties window
tartup project
ollapse tree
xpand tree
Solution Explorer - WindowsApplication1
Solution 'WindowsApplication1' (1 project)
WindowsApplication1
References
App.ico
AssemblyInfo.cs
Form1.cs
Solution Explorer
Class View
Solution Explorer
Solution Explorer
Class View

Solution Explorer with an open solution

Solution Explorer without an open solution

Fig. 2.9 **Solution Explorer** window.

The **Solution Explorer** contains a toolbar. One icon on the toolbar reloads the files in the solution (refreshes), and another icon displays all files in the solution (including hidden ones). The number of icons in the toolbar changes depending on the type of file selected. We discuss these icons later in the book.

2.4.2 Toolbox

The **Toolbox** (Fig. 2.10) contains reusable software components (or controls) that can be used to customize applications. Using *visual programming*, programmers can “drag and drop” controls onto a form instead of writing code themselves. Just as people do not need to know how to build an engine to drive a car, programmers do not need to build a control to use it. This allows them to concentrate on the big picture, rather than the complex details of every control. The wide variety of tools available to programmers is a powerful feature of C#. We demonstrate the power of the controls in the **Toolbox** when we create our own program later in the chapter.

The **Toolbox** contains groups of related components (e.g., **Data**, **Components**, **Windows Forms**). Expand the members of a group by clicking the name of the group. Users can scroll through the individual items by using the black scroll arrows on the right side of the **Toolbox**. The first item in the group is not a control—it is the mouse pointer. Clicking this icon allows the user to deselect the current control in the **Toolbox**. Note that there are no tool tips, because the **Toolbox** icons already are labeled with the names of the controls. In later chapters, we discuss many of these controls.

Initially, the **Toolbox** may be hidden, with only the name of the window showing on the side of the IDE (Fig. 2.11). Moving the mouse pointer over a window name opens this window. Moving the mouse pointer outside the window causes the window to disappear. This feature is known as *auto hide*. To “pin down” the **Toolbox** (i.e., disable auto hide), click the pin icon in the upper right corner of the window (see Fig. 2.11). To enable auto hide (if it previously has been disabled), click the pin icon again. Notice that when auto hide is enabled, the pin points to the side, as is shown in Fig. 2.11.

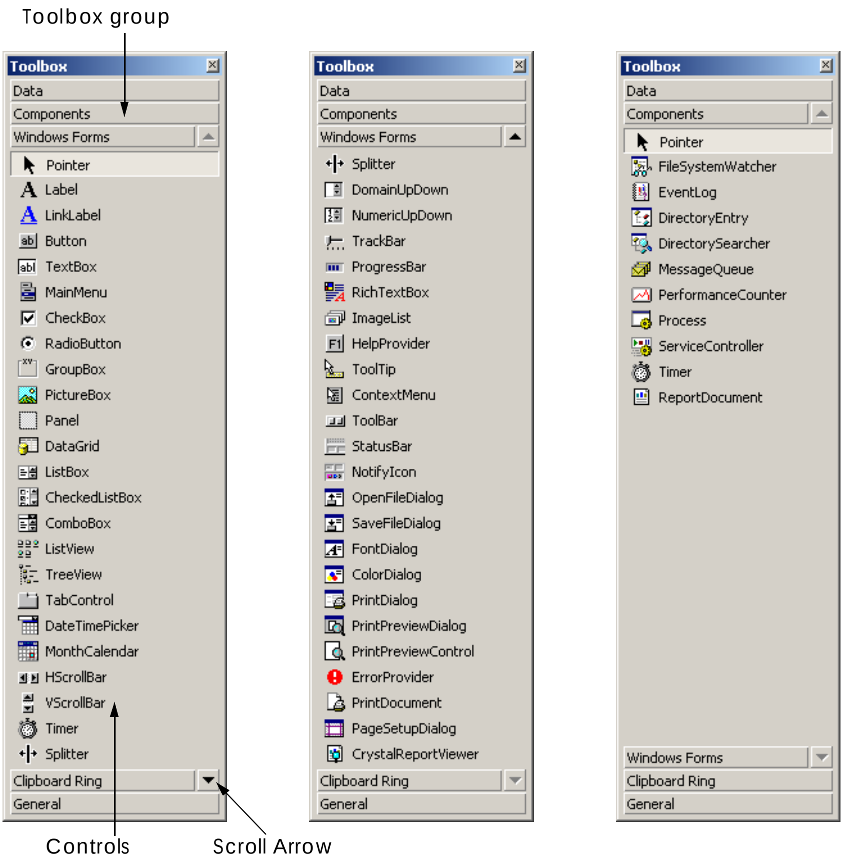

Toolbox group
Toolbox
Data
Components
Windows Forms
Pointer
Label
LinkLabel
Button
TextBox
MainMenu
CheckBox
RadioButton
GroupBox
PictureBox
Panel
DataGrid
ListBox
CheckedListBox
ComboBox
ListView
TreeView
TabControl
DateTimePicker
MonthCalendar
HScrollBar
VScrollBar
Timer
Splitter
Clipboard Ring
General
Toolbox
Data
Components
Windows Forms
Splitter
DomainUpDown
NumericUpDown
TrackBar
ProgressBar
RichTextBox
ImageList
HelpProvider
ToolTip
ContextMenu
ToolBar
StatusBar
NotifyIcon
OpenFileDialog
SaveFileDialog
FontDialog
ColorDialog
PrintDialog
PrintPreviewDialog
PrintPreviewControl
ErrorProvider
PrintDocument
PageSetupDialog
CrystalReportViewer
Clipboard Ring
General
Toolbox
Data
Components
Pointer
FileSystemWatcher
EventLog
DirectoryEntry
DirectorySearcher
MessageQueue
PerformanceCounter
Process
ServiceController
Timer
ReportDocument
Windows Forms
Clipboard Ring
General
Controls
Scroll Arrow

Fig. 2.10 **Toolbox** window.

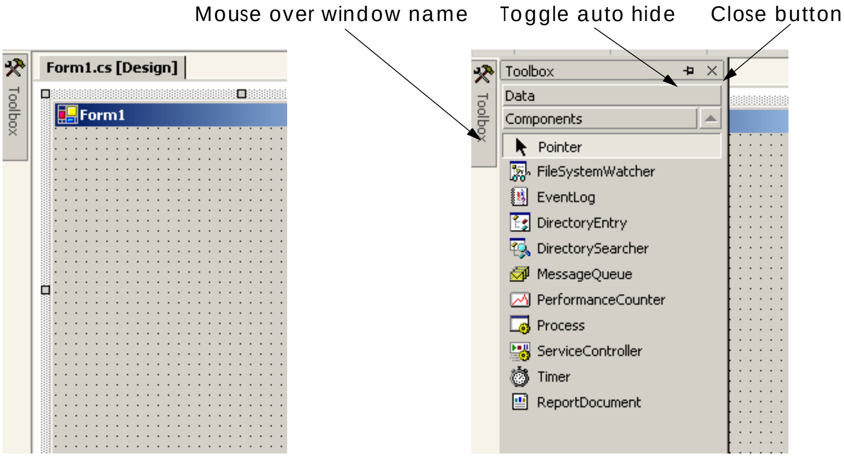

Mouse over window name
Toggle auto hide
Close button
Toolbox
Form1.cs [Design]
Form1
Toolbox
Toolbox
Data
Components
Pointer
FileSystemWatcher
EventLog
DirectoryEntry
DirectorySearcher
MessageQueue
PerformanceCounter
Process
ServiceController
Timer
ReportDocument

Fig. 2.11 Demonstrating window auto-hide.

2.4.3 Properties Window

The **Properties** window (Fig. 2.12) allows manipulation of the *properties* for a form or control. Properties specify information about a control, such as size, color and position. Each control has its own set of properties. The bottom of the **Properties** window contains a description of the selected property.

The left column of the **Properties** window shows the properties of the control (a form in Fig. 2.12). The right column displays their current values. Icons on the toolbar sort the properties either alphabetically (by clicking the ***Alphabetic*** *icon*) or categorically (by clicking the ***Categorized*** *icon*). Users can scroll through the list of properties by *dragging* the scrollbar up or down (i.e., holding down the left mouse button while the mouse cursor is over the scrollbar, moving the mouse up or down and releasing the mouse button). The ***Event*** *icon* allows the control or form to respond to certain user actions. We discuss events in Chapter 12, Graphical User Interface Concepts: Part 1. We show how to set individual properties later in this chapter and throughout the book.

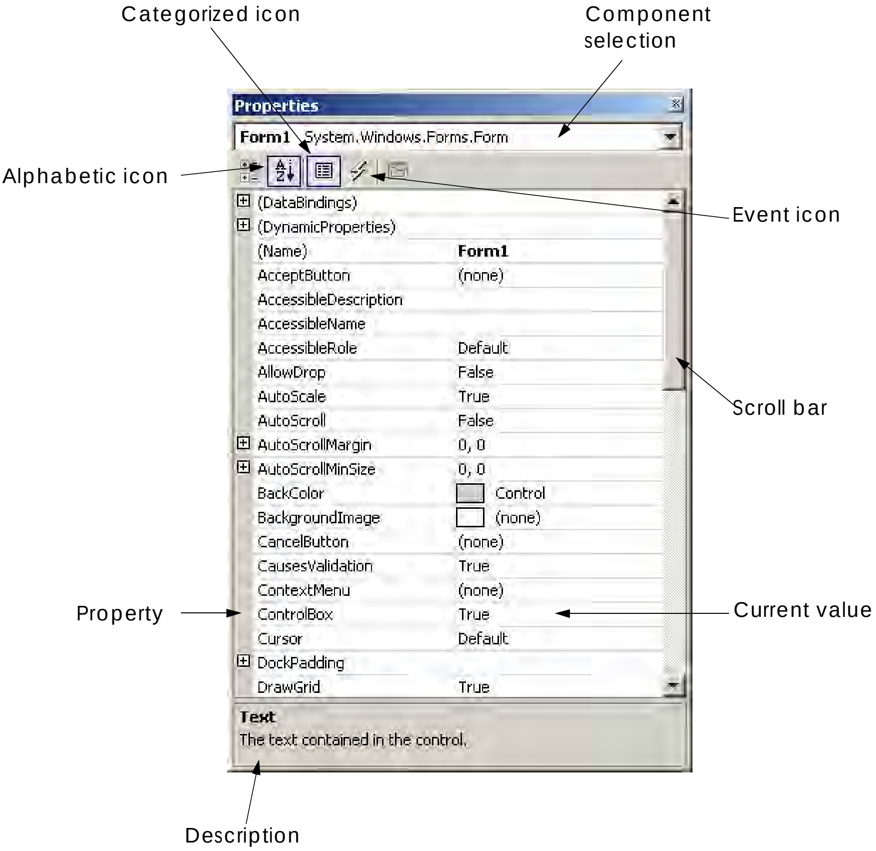
The **Properties** window also is important to visual programming. Controls are usually customized after they are created from the **Toolbox**. The **Properties** window allows programmers to modify controls visually, without writing code. This setup has a number of benefits. First, the programmer can see which properties are available for modification and what the possible values are; the programmer does not have to look up or remember what settings a particular property can have. Second, the window displays a brief description of each property, allowing the programmer to understand each property's purpose. Third, a property's value can be set quickly using the window; only a single click is required, and no code need be written. All these features are designed to help software developers program without performing many repetitive tasks.

At the top of the **Properties** window is a drop-down list called the *component selection*. This list shows the current component that is being altered. The programmer can use the list to choose which component to edit. For example, if a GUI contains several buttons, the programmer can select the name of a specific button to configure.

2.5 Using Help

Visual Studio .NET has an extensive help mechanism. The ***Help*** *menu* contains a variety of options. The ***Contents*** *menu item* displays a categorized table of contents. Menu item ***Index*** displays an alphabetical index that users can browse. The **Search** feature allows users to find particular help articles based on a few search words. In each case, a filter can narrow the search to articles related only to C#.

Dynamic help (Fig. 2.13) provides a list of articles based on the current content (i.e., the items around the location of the mouse cursor). To open dynamic help (if it is not already open), select the **Help** menu's **Dynamic Help** command. Once you click an object to display in Visual Studio .NET, relevant help articles will appear in the **Dynamic Help** window. The window lists relevant help entries, samples and "Getting Started" information, in addition to providing a toolbar for the regular help features. Dynamic help is an excellent way to get information about the features of Visual Studio .NET. Note that for some users, **Dynamic Help** slows down Visual Studio.



Properties

Form1 System.Windows.Forms.Form

⊞ (DataBindings)	
⊞ (DynamicProperties)	
(Name)	**Form1**
AcceptButton	(none)
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
AutoScale	True
AutoScroll	False
⊞ AutoScrollMargin	0, 0
⊞ AutoScrollMinSize	0, 0
BackColor	▭ Control
BackgroundImage	☐ (none)
CancelButton	(none)
CausesValidation	True
ContextMenu	(none)
ControlBox	True
Cursor	Default
⊞ DockPadding	
DrawGrid	True

Text
The text contained in the control.

Fig. 2.12 **Properties** window.

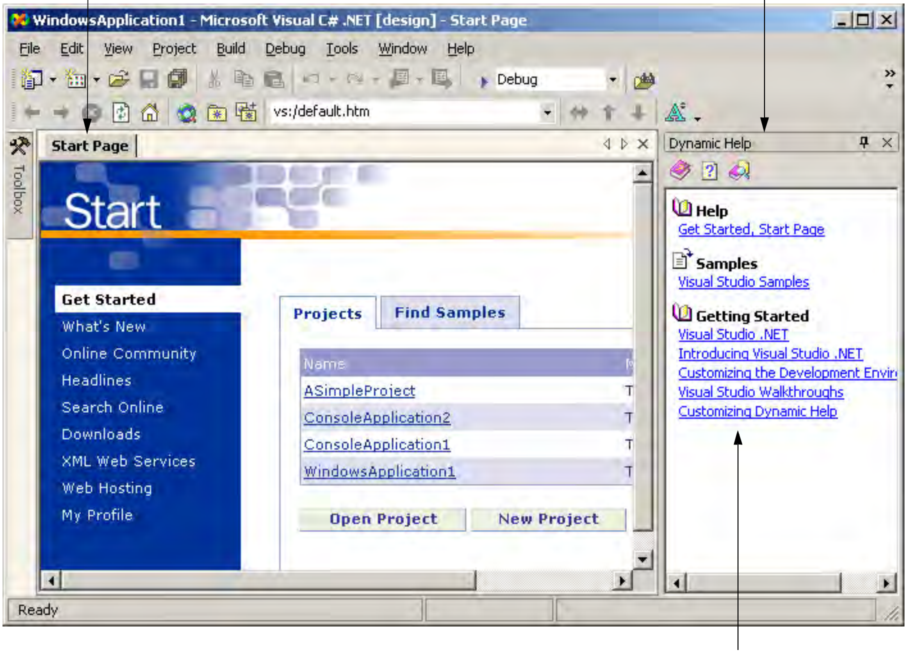
Performance Tip 2.1

If you experience slow response times from Visual Studio, you can disable (i.e., close) ***Dynamic Help*** *by clicking the* **X** *in the upper-right corner of the window.*

In addition to dynamic help, Visual Studio .NET provides *context-sensitive help*. Context-sensitive help is similar to dynamic help, except that context-sensitive text immediately brings up a relevant help article rather than presenting a list. To use context-sensitive help, select an item and press the *F1* key. Help can appear either *internally* or *externally*. With external help, a relevant article immediately pops up in a separate window, outside the IDE. With internal help, a help article appears as a tabbed window inside Visual Studio .NET. The help options can be set from the **My Profile** section of the **Start Page**. Dynamic help and context-sensitive help are explained in the context of C# code later in the book.

Selected item

Dynamic Help window


WindowsApplication1 - Microsoft Visual C# .NET [design] - Start Page
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug
vs:/default.htm
Toolbox
Start Page
Start
Get Started
What's New
Online Community
Headlines
Search Online
Downloads
XML Web Services
Web Hosting
My Profile
Projects
Find Samples
Name
ASimpleProject
ConsoleApplication2
ConsoleApplication1
WindowsApplication1
Open Project
New Project
Dynamic Help
Help
Get Started, Start Page
Samples
Visual Studio Samples
Getting Started
Visual Studio .NET
Introducing Visual Studio .NET
Customizing the Development Envir
Visual Studio Walkthroughs
Customizing Dynamic Help
Ready

Relevant help articles for currently selected item (**Start Page**)

Fig. 2.13 **Dynamic Help** window.

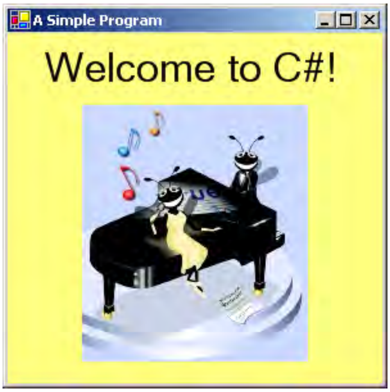
2.6 Simple Program: Displaying Text and an Image

In this section, we create a program that displays the text “`Welcome to C#!`” and an image. The program consists of a single form that uses a label to display text and a picture box to display an image. Figure 2.14 shows the program as it executes. The example here (as well as the image file used in the example) is available on our Web Site (`www.deitel.com`) under the **Downloads/Resources** link.

We do not write a single line of program code. Instead, we use the techniques of visual programming. Various programmer *gestures* (such as using the mouse for pointing, clicking, dragging and dropping) provide Visual Studio .NET with sufficient information for it to generate all or a major portion of the program code. In the next chapter, we begin our discussion of writing program code. Throughout the book, we produce increasingly substantial and powerful programs. Visual C# programming usually involves a combination of writing a portion of the program code and having Visual Studio .NET generate the remaining code.

To create, run and terminate this first program, perform the following steps:

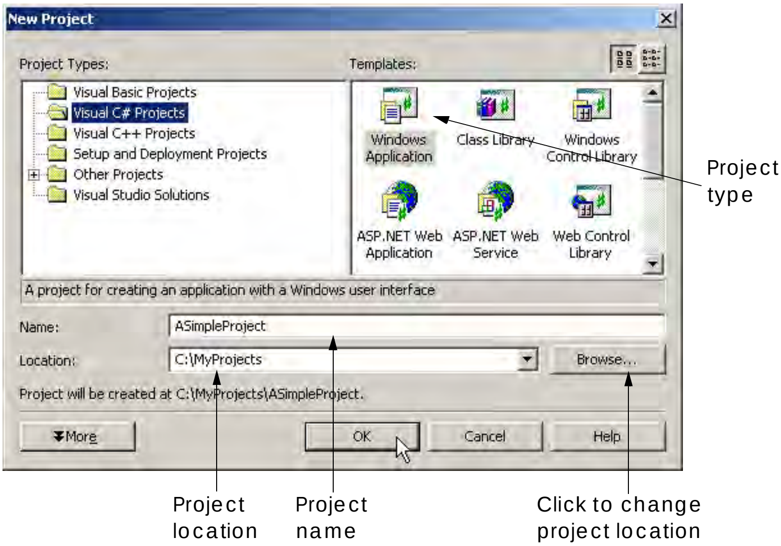
1. *Create the new project.* If a project is already open, close it by selecting **File > Close Solution** from the menu. A dialog asking whether to save the current solution may appear in order to keep any unsaved changes, save the solution. Create


A Simple Program

Welcome to C#!

Fig. 2.14 Simple program as it executes.

a new Windows application for our program. Open Visual Studio .NET, and select **File > New > Project... > Visual C# Projects > Windows Application** (Fig. 2.15). Name the project **`ASimpleProgram`**, and select a directory in which to save the project. To do this, click the **Browse...** button, which opens a **Project Location** dialog (Fig. 2.16). Navigate through the directories, find one in which to place the project and select **OK**. This selection returns us to the **New Project** dialog; the selected folder appears in the **Location** text field. When you are satisfied with the location of the project, click **OK**. Visual Studio .NET will load the new solution, and a form labeled **Form1** will appear.


New Project
Project Types:
Visual Basic Projects
Visual C# Projects
Visual C++ Projects
Setup and Deployment Projects
Other Projects
Visual Studio Solutions
Templates:
Windows Application
Class Library
Windows Control Library
ASP.NET Web Application
ASP.NET Web Service
Web Control Library
More
OK
Cancel
Help
Browse...
Project type
Project location
Project name
Click to change project location

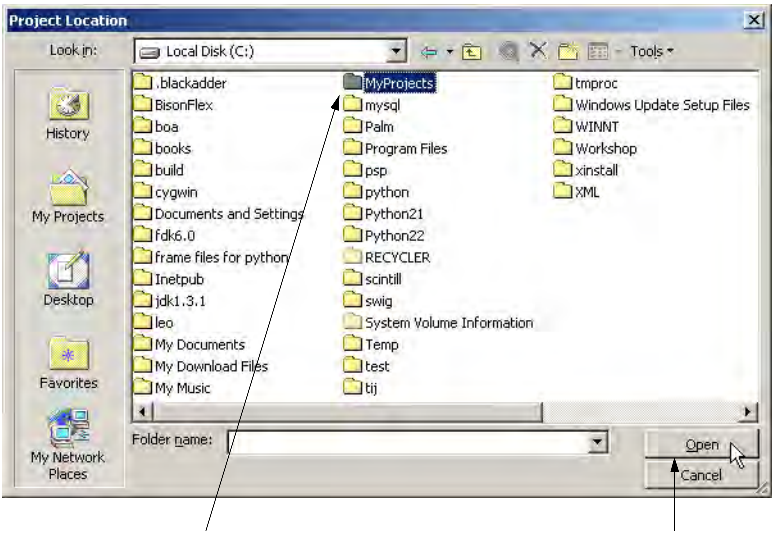
A project for creating an application with a Windows user interface

Name: ASimpleProject

Location: C:\MyProjects

Project will be created at C:\MyProjects\ASimpleProject.

Fig. 2.15 Creating a new Windows application.

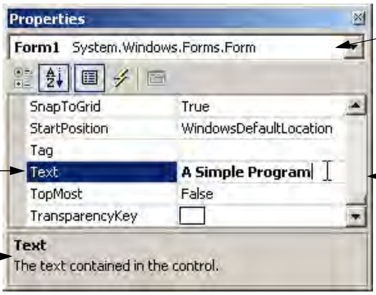

Project Location
Look in:
Local Disk (C:)
Tools
History
My Projects
Desktop
Favorites
My Network Places
.blackadder
BisonFlex
boa
books
build
cygwin
Documents and Settings
fdk6.0
frame files for python
Inetpub
jdk1.3.1
leo
My Documents
My Download Files
My Music
MyProjects
mysql
Palm
Program Files
psp
python
Python21
Python22
RECYCLER
scintill
swig
System Volume Information
Temp
test
tij
tmproc
Windows Update Setup Files
WINNT
Workshop
xinstall
XML
Folder name:
Open
Cancel

Selected project location

Click to set project location

Fig. 2.16 Setting the project location.

2. *Set the form's title bar*. First, set the text that appears in the title bar. This text is determined by the form's ***`Text`*** property (Fig. 2.17). If the form's **Properties** window is not open, click the **Properties** icon in the toolbar or select the **View** menu's **Properties Window** command. Use the mouse to select the form; the **Properties** window shows information about the currently selected item. In the window, click in the box to the right of the **`Text`** property's box. To set a value for the **`Text`** property, type the value in the box. In this case, type **`A Simple Program`**, as in Fig. 2.17. Press the *Enter* key (the *Return* key) when you have finished to update the form's title bar in the design area.


Properties
Form1 System.Windows.Forms.Form

SnapToGrid	True
StartPosition	WindowsDefaultLocation
Tag	
Text	A Simple Program
TopMost	False
TransparencyKey	

Text
The text contained in the control.

Name and type of object

Selected property

Property value

Property description

Fig. 2.17 Setting the form's **`Text`** property.

3. *Resize the form*. Click and drag one of the form's enabled *sizing handles* (the small squares around the form shown in Fig. 2.18) to change the size of the form. Enabled sizing handles are white. The mouse cursor changes appearance when it is over an enabled sizing handle. Disabled sizing handles are gray. The grid on the background of the form is used to align controls and does not appear when the program executes.
4. *Change the form's background color*. The ***`BackColor`*** property specifies a form's or control's background color. Clicking **`BackColor`** in the **Properties** window causes a down-arrow button to appear next to the property value (Fig. 2.19). When clicked, the down arrow drops down to display other options. (The options vary, depending on the property.) In this case, it displays the tabs ***System*** (the default), ***Web*** and ***Custom***. Click the **Custom** tab to display the *palette* (a selection box of colors). Select the box that represents yellow. The palette will disappear, and the form's background color will change to yellow.

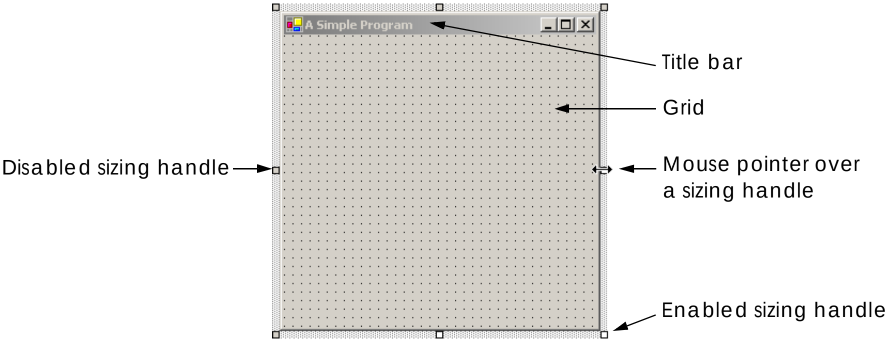
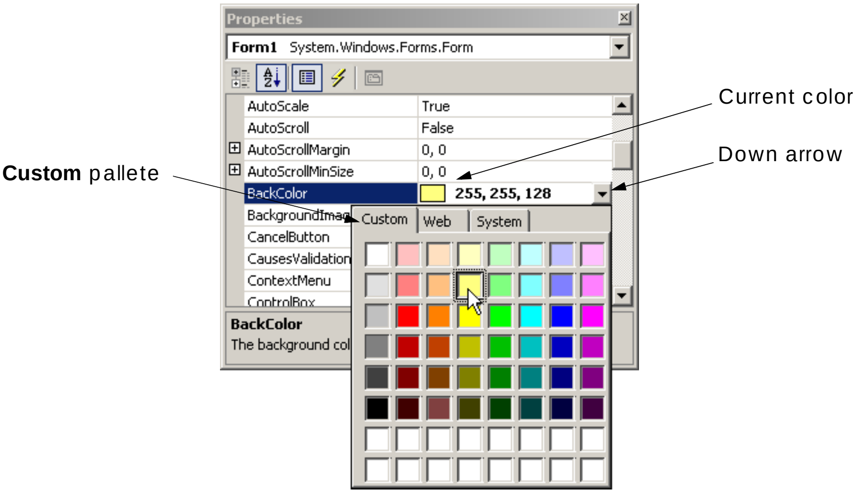

A Simple Program
Title bar
Grid
Disabled sizing handle
Mouse pointer over a sizing handle
Enabled sizing handle

Fig. 2.18 Form with sizing handles.


Properties
Custom
Web
System
Current color
Down arrow
Custom pallete

Form1 System.Windows.Forms.Form

AutoScale	True
AutoScroll	False
⊞ AutoScrollMargin	0, 0
⊞ AutoScrollMinSize	0, 0
BackColor	🟨 255, 255, 128
BackgroundImag	
CancelButton	
CausesValidation	
ContextMenu	
ControlBox	

BackColor
The background col

Fig. 2.19 Changing property **`BackColor`**.

5. *Add a label control to the form*. Double-click the label control in the **Toolbox**. This action creates a label with sizing handles in the upper-left corner of the form (Fig. 2.20). Double-clicking any **Toolbox** control places it on the form. Alternatively, programmers can "drag" controls from the **Toolbox** to the form. Labels display text; our label displays **`label1`** by default. Notice that our label is the same color as the form's background color. The form's background color is also the default background color of controls added to the form.

6. *Set the label's text*. Select the label so that its properties appear in the **Properties** window. The label's **`Text`** property determines the text (if any) that the label displays. The form and label each have their own **`Text`** property. Forms and controls can have the same types of properties without conflict. We will see that many controls have property names in common. Set the **`Text`** property of the label to **`Welcome to C#!`** (Fig. 2.21). Resize the label (using the sizing handles) if the text does not fit. Move the label to the top center of the form by dragging it or using the arrow keys. Alternatively, you can move the label by selecting **Format > Center In Form > Horizontally** from the menu bar.

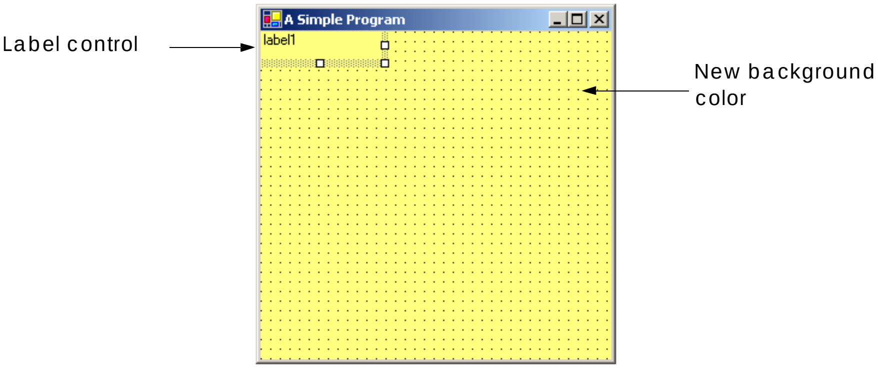


Fig. 2.20 Adding a new label to the form.

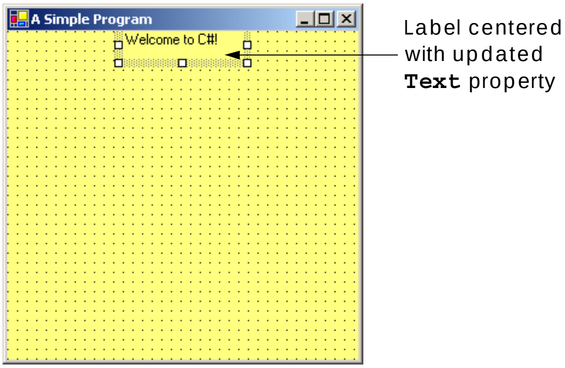
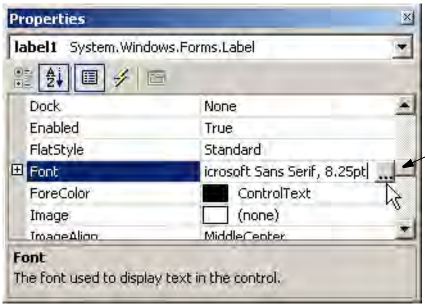


Fig. 2.21 Label in position with its `Text` property set.

7. *Set the label's font size, and align the label's text*. Clicking the **`Font`** property value causes an *ellipsis* button (**...**) to appear next to the value, as in Fig. 2.22. The ellipsis button indicates that a dialog will appear when the programmer clicks the button. When the button is clicked, the ***Font*** *window* shown in Fig. 2.23 is displayed. Users can select the font name (**Microsoft Sans Serif**, **Arial**, etc.), font style (**Regular**, **Bold**, etc.) and font size (**8**, **10**, etc.) in this window. The text in the ***Sample*** *area* displays the selected font. Under the **Size** category, select **24** and click **OK**. If the text does not fit on a single line, it will wrap to the next line. Resize the label if it is not large enough to hold the text. Next, select the label's ***`TextAlign`*** property, which determines how the text is aligned within the label. A three-by-three grid of alignment choices is displayed, corresponding to where the text appears in the label (Fig. 2.24). Select the top-center grid item, so that the text will appear at the top center of the label.

8. *Add a picture box to the form*. The picture-box control displays images. This step is similar to Step 5. Find the picture box in the toolbox, and add it to the form. Move it underneath the label, by either dragging it or using the arrow keys (Fig. 2.25).



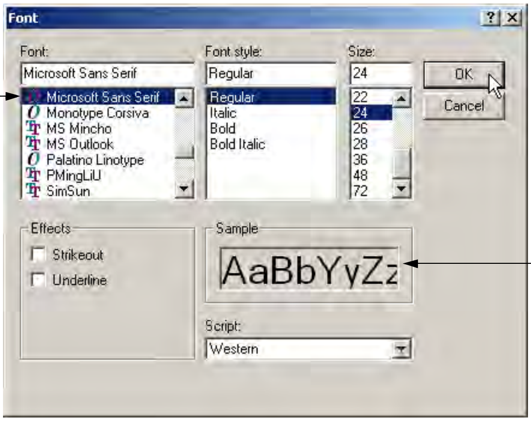
label1 System.Windows.Forms.Label

Dock	None
Enabled	True
FlatStyle	Standard
Font	icrosoft Sans Serif, 8.25pt
ForeColor	ControlText
Image	(none)
ImageAlign	MiddleCenter

Font
The font used to display text in the control.

Ellipsis indicates dialog will appear

Fig. 2.22 **Properties** window displaying the label's properties.



Font: Microsoft Sans Serif

Microsoft Sans Serif
Monotype Corsiva
MS Mincho
MS Outlook
Palatino Linotype
PMingLiU
SimSun

Font style: Regular

Regular
Italic
Bold
Bold Italic

Size: 24

22
24
26
28
36
48
72

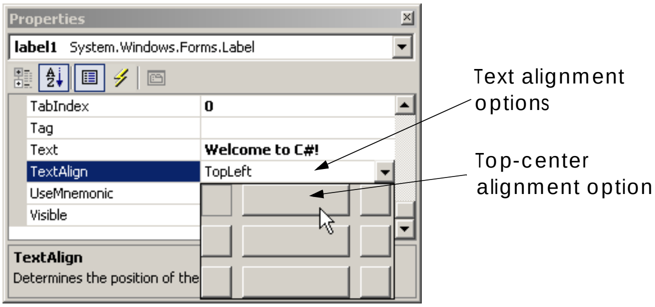
☐ Strikeout
☐ Underline

Script: Western

Current font

Font sample

Fig. 2.23 **Font** window for selecting fonts, styles and sizes.



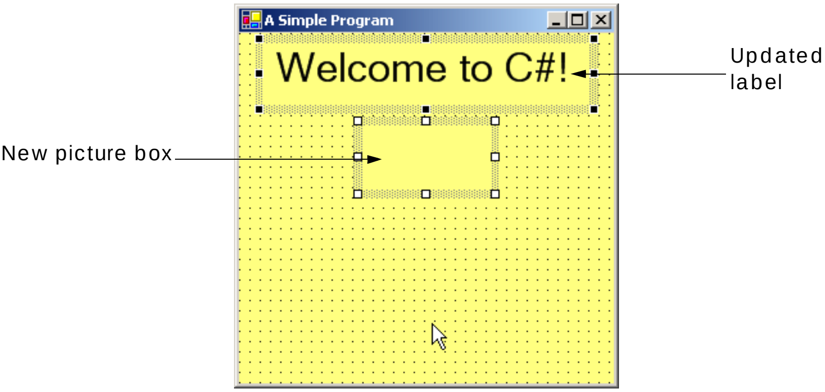
TabIndex 0
Tag
Text **Welcome to C#!**
TextAlign TopLeft
UseMnemonic
Visible

TextAlign
Determines the position of the

Text alignment options

Top-center alignment option

Fig. 2.24 Centering the text in the label.

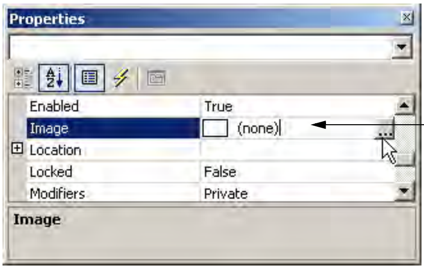


Updated label

New picture box

Fig. 2.25 Inserting and aligning the picture box.

9. *Insert an image*. Click the picture box to load its properties in the **Properties** window, and find the ***`Image`*** *property*. The **`Image`** property shows a preview of the current picture. No picture has been assigned, so the **`Image`** property displays **(none)** (Fig. 2.26). Click the ellipsis button to display an **Open** dialog (Fig. 2.27). Browse for a picture to insert, and press *Enter* key. The proper formats of an image include PNG (Portable Networks Graphic), GIF (Graphic Interchange Format) and JPEG (Joint Photographics Experts Group). Each of these file formats is widely supported on the Internet. To create a new picture, it is necessary to use image-editing software, such as Jasc Paint Shop Pro, Adobe Photoshop Elements or Microsoft Paint. We use the picture **`ASimpleProgramImage.png`**, which is located with this example on the CD that accompanies the book and on our Web site (**`www.deitel.com`**). After the image has been inserted, the picture box displays as much of the picture as it can (depending on size) and the **`Image`** property shows a small preview. To display the entire image, resize the picture box by dragging the picture box's handles (Fig. 2.28).

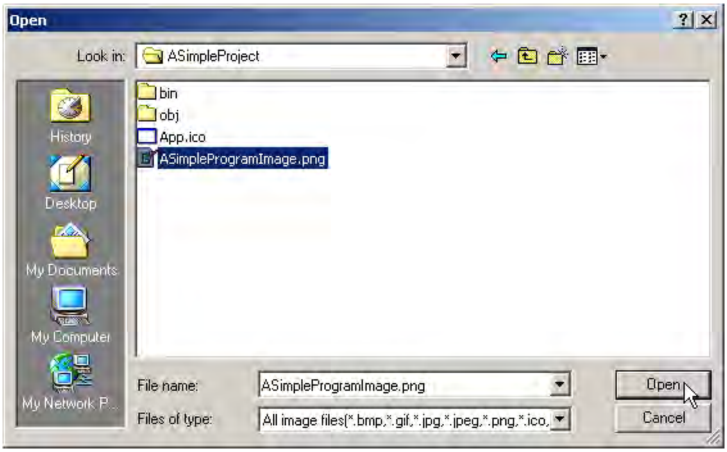

Properties

Enabled	True
Image	(none)
⊞ Location	
Locked	False
Modifiers	Private

Image

Image property value (no image selected)

Fig. 2.26 `Image` property of the picture box.


Open
History
Desktop
My Documents
My Computer
My Network P...
Open
Cancel

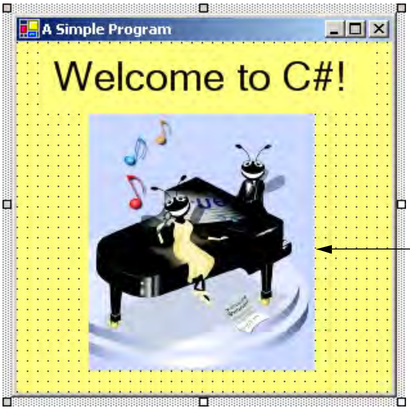
Look in: ASimpleProject

bin
obj
App.ico
ASimpleProgramImage.png

File name: ASimpleProgramImage.png

Files of type: All image files(*.bmp,*.gif,*.jpg,*.jpeg,*.png,*.ico,

Fig. 2.27 Selecting an image for the picture box.

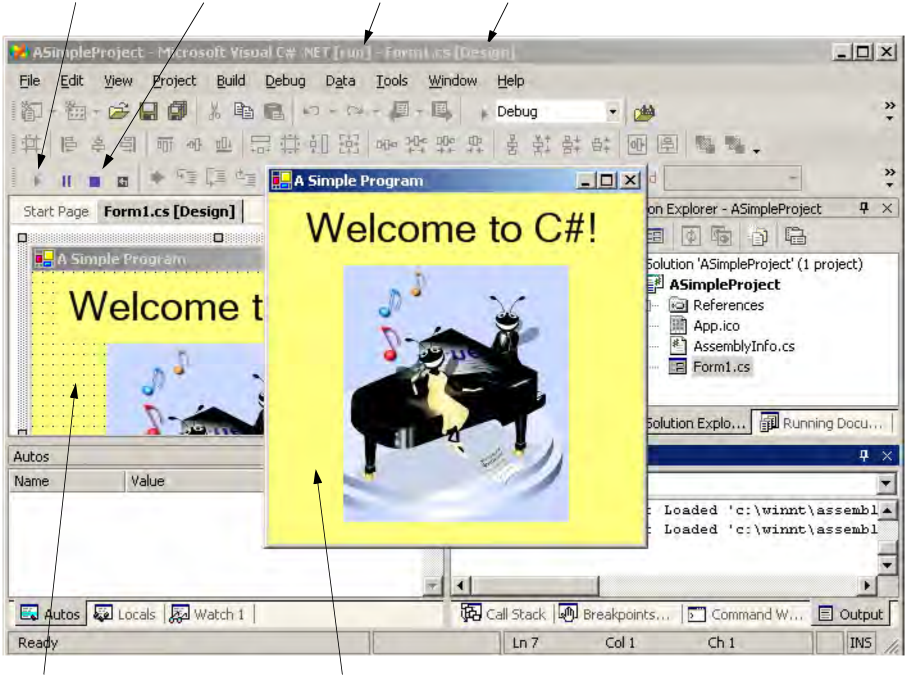

A Simple Program
Welcome to C#!

Newly inserted image (after resizing the picture box)

Fig. 2.28 Picture box after the image has been inserted.

10. *Save the project*. Select **File > Save All** to save the entire solution. To save an individual file, select it in the **Solution Explorer**, and select **File > Save**. The created program stores the source code in the C# file `Form1.cs`. The project file contains the names and locations of all the files in the project. Choosing **Save All** saves both the project and the C# file.
11. *Run the project*. Prior to this step, we have been working in the IDE *design mode* (i.e., the program being created is not executing). This mode is indicated by the text **Microsoft Visual C# .NET [design]** in the title bar. While in design mode, programmers have access to all the environment windows (i.e., **Toolbox** and **Properties**), menus, toolbars and so forth. While in *run mode*, however, the program is executing, and users can interact with only a few IDE features. Features that are not available are disabled or grayed out. The text **Form1.cs [Design]** in the title bar means that we are designing the form visually, rather than programming it using code. If we had been writing code, the title bar would have contained only the text **Form1.cs**. To execute or run our program, we first need to compile it, which is accomplished by clicking on the ***Build Solution*** *option* in the ***Build*** *menu* (or type *<Ctrl> + Shift + B*). The program can then be executed by clicking the ***Start*** button (the blue triangle), selecting the **Debug** menu's **Start** command or pressing the *F5* key. Figure 2.29 shows the IDE in run mode. Note that the IDE title bar displays **[run]** and that many toolbar icons are disabled.

Start button End button Run mode Designing form


ASimpleProject - Microsoft Visual C# .NET [run] - Form1.cs [Design]
File
Edit
View
Project
Build
Debug
Data
Tools
Window
Help
Debug
A Simple Program
Welcome to C#!
Start Page
Form1.cs [Design]
A Simple Program
Welcome t
Solution Explorer - ASimpleProject
Solution 'ASimpleProject' (1 project)
ASimpleProject
References
App.ico
AssemblyInfo.cs
Form1.cs
Solution Explo...
Running Docu...
Autos
Name
Value
Autos
Locals
Watch 1
Call Stack
Breakpoints...
Command W...
Output
Ready
Ln 7
Col 1
Ch 1
INS

```
Loaded 'c:\winnt\assembl
Loaded 'c:\winnt\assembl
```

Form design (with grid) Running application

Fig. 2.29 IDE in run mode, with the running application in the foreground.

12. *Terminating execution*. To terminate the program, click the running application’s ***Close*** button (the **x** in the top-right corner). Alternatively, click the ***End*** button (the blue square) in the toolbar. Either action stops program execution and puts the IDE into design mode.

We have just created a working C# program without writing a single line of code. Visual programming allows us to create controls and set properties using windows, rather than lines of code. In the next chapter, we discuss nonvisual, or conventional, programming—we create a program using only code. C# programming is a mixture of the two styles: Visual programming allows us to develop a GUI and avoid tedious tasks, while conventional programming specifies the behavior of our program. The most important part of an application is its behavior, which we explain how to program in the upcoming chapters.

Software Engineering Observation 2.1

Visual programming can be simpler and faster than writing code.

Software Engineering Observation 2.2

Most programs require more than visual programming. In such programs, some code must be written by hand. Examples include applications that use event handlers (used to respond to the user’s actions), databases, security, networking, text editing, graphics and multimedia.

SUMMARY

- Visual Studio .NET is Microsoft’s integrated development environment (IDE) for creating, documenting, running and debugging programs.
- When Visual Studio .NET is loaded for the first time, the **Start Page** is displayed. This page contains helpful links, such as recent projects, online newsgroups, downloads and user profile settings.
- The **Get Started** section contains links to recent files.
- The **My Profile** page allows users to customize Visual Studio .NET.
- In the Visual Studio .NET IDE, users can browse the Web via Internet Explorer.
- Dialogs are windows that are used to communicate with users.
- Programs in Visual Studio .NET are organized into projects and solutions. A project is a group of related files. A solution is a group of projects that are combined to solve a developer’s problem.
- Windows applications are programs that execute inside the Windows OS, like Microsoft Word, Internet Explorer and Visual Studio .NET. They contain controls—reusable graphical elements, such as buttons and labels—which the user uses to interact with the application.
- The form is what the users interact with and view when programs run.
- The form and its controls constitute the graphical user interface (GUI) of the program. Controls are the graphical components with which the user interacts. Users enter data (inputs) into the program by entering information from the keyboard and clicking the mouse buttons. The program displays instructions and other information (outputs) for users to read in the GUI.
- The title bar displays the name of the project, the programming language, the mode of the IDE, the file being viewed and the mode of the file being viewed.
- To view a tabbed document, click the tab with the name of the document. Tabbing saves space and allows easy access to multiple documents.

- Menus contain groups of related commands that, when selected, cause the IDE to perform some action. Visual Studio .NET provides different modes for the user. Certain menu items appear only in some of these modes.
- The toolbar contains icons that represent menu commands. To execute a command, click the corresponding icon. Click the down arrow beside an icon to display other available options.
- Moving the mouse pointer over an icon highlights the icon and displays a tool tip.
- The **Solution Explorer** window lists all the files in the solution.
- The startup project of the solution is the project that runs when the program is executed.
- The plus and minus boxes to the left of the project and solution names expand and collapse the tree, respectively.
- The **Toolbox** contains controls that customize forms.
- By using visual programming, programmers can “drag and drop” controls onto the form instead of writing the code themselves.
- Moving the mouse pointer over the label of a hidden window opens the window. Moving the mouse pointer outside the window causes the window to disappear. This feature is known as auto hide. To “pin down” the **Toolbox** window (i.e., to disable auto hide), click the pin icon in the upper-right corner.
- The **Properties** window displays the properties for a form or control. Properties are information about a control, such as size, color and position.
- Each type of control has its own set of properties.
- The left column of the **Properties** window shows the properties of the control. The right column displays their current values. The toolbar sorts the properties either alphabetically (by clicking the **Alphabetic** icon) or categorically (by clicking the **Categorized** icon).
- The **Properties** window allows programmers to modify controls visually, without writing code.
- The **Help** menu contains a variety of options. The **Contents** menu item displays a categorized table of contents. Menu item **Index** displays an alphabetical index that can be browsed. The **Search** feature allows users to find particular help articles, based on a few search words.
- For each option of the **Help** menu, a filter can be used to narrow the search to articles relating only to C#.
- Dynamic help provides a list of articles, based on the current content (i.e., the location of the mouse cursor).
- Context-sensitive help is similar to dynamic help, except that context-sensitive help immediately brings up a relevant help article. To use context-sensitive help, select an item and press the *F1* key.
- Visual C# programming usually involves a combination of writing a portion of the program code and having Visual Studio .NET generate the remaining code.
- To create a new Windows Forms project, open Visual Studio .NET and select **File > New > Project...> Visual C# Projects > Windows Application**. Name the project, and select a directory. Then click **OK**. Visual Studio .NET will load the new solution, and a blank form labeled **Form1** will appear.
- The text that appears on the top of the form (the title bar) is determined by the `Text` property of the form. To set a value for the property, simply type it in the space provided. Press the *Enter* key (*Return* key) when you have finished.
- To resize the form, click and drag one of the form’s enabled sizing handles (the small squares around the form). Enabled sizing handles are white; disabled sizing handles are gray.
- The grid on the background of the form is used to align controls and does not appear when the program is running.

- The `BackColor` property specifies a form's or control's background color. The form's background color is the default background color for any controls added to the form.
- Double-clicking any **Toolbox** control icon places a control of that type on the form. Alternatively, programmers can "drag" controls from the **Toolbox** to the form.
- The label's `Text` property determines the text (if any) that the label displays. The form and label each have their own `Text` property.
- When clicked, the ellipsis button displays a dialog.
- In the **Font** dialog users can select a font using the font name, font style and font size.
- The `TextAlign` property determines how the text is aligned within the label's boundaries.
- The picture-box control allows us to display an image on the form. The `Image` property shows a preview of the current picture. To select an image, click the ellipsis button, which displays an **Open** dialog. Browse for a picture to insert (of the proper format, such as PNG, GIF or JPEG), and then press the *Enter* key.
- Select **File > Save All** to save the entire solution. To save an individual file, select it in the **Solution Explorer** and select **File > Save**.
- The IDE design mode (i.e., the program is not executing) is indicated by the text **Microsoft Visual C# .NET [Design]** in the title bar.
- While in run mode, the program is executing, and users can interact with only a few IDE features.
- When designing a program visually, the name of the C# file will appear in the title bar, followed by **[Design]**.
- To execute or run a program, click the **Start** button (the blue triangle), or select **Debug> Start**. The IDE title bar displays **[Run]**, and many toolbar icons are disabled.
- Terminate execution by clicking the **Close** button. Alternatively, click the **End** button (a blue square) in the toolbar.

TERMINOLOGY

`Alignment` property
Alphabetic icon
Appearance category in the **Properties** window
auto hide
`BackColor` property
background color
Build menu
button
Categorized icon
clicking
close a project
Close button icon
collapse a tree
compile a program
context-sensitive help
control
control layout
customize a form
customize Visual Studio .NET
Data menu
debug a program
Debug menu
design mode
dialog
double-clicking
down arrow
dynamic help
Dynamic Help window
Edit menu
expand a tree
external help
F1 help key
File menu
find
`Font` property
font size
font style
Font window
form
form's background color
form's title bar

Format menu
GUI (Graphical User Interface)
help filter
Help menu
icon
IDE (integrated development environment)
input
internal help
Internet Explorer
label
menu
menu bar in Visual Studio .NET
mouse pointer
new project in Visual Studio .NET
opening a project
output
palette
paste
picture box
pin a window
print a project
project
Project menu
Properties window
property
property for a form or control
recent project
Run menu
run mode
selecting
single-clicking with the left mouse button
sizing handle
solution
Solution Explorer in Visual Studio .NET
Start button
Start Page
startup project
tabbed window
`Text` property
title bar
tool tip
toolbar
toolbar icon
Tools menu
undo
View menu
Visual Studio .NET
window layout
Windows application
Windows menu

SELF-REVIEW EXERCISES

2.1 Fill in the blanks in each of the following statements:

a) The technique of __________ allows us to create a GUI without writing any code.
b) A __________ is a group of related files, compiled into one application.
c) The __________ feature saves screen space when the mouse is moved away from a window.
d) A __________ appears when the mouse cursor hovers over an icon.
e) The __________ window allows you to browse the files in your solution.
f) A plus icon indicates that the tree in the **Solution Explorer** can __________.
g) The **Properties** window can be sorted __________ or __________.
h) The form’s __________ property determines the text that appears in its title bar.
i) The __________ allows us to add controls to the form visually.
j) __________ displays relevant help articles, based on the current context.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The title bar displays the mode of the IDE.
b) The **Start Page** allows the user to customize the IDE.
c) The **x** button toggles auto hide in most windows.
d) The toolbar provides a convenient way to execute certain menu commands.
e) The toolbar contains the control icons.
f) A form’s sizing handles are always enabled when the form is selected.
g) Both forms and labels have a title bar.
h) Controls can be modified only by writing code.
i) Buttons usually perform actions when clicked.
j) The grid appears when designing a form, but not during execution.

ANSWERS TO SELF-REVIEW EXERCISES

2.1 a) visual programming. b) project. c) auto hide. d) tool tip. e) **Solution Explorer**. f) expand. g) alphabetically, categorically. h) **`Text`**. i) **Toolbox**. j) Dynamic help.

2.2 a) True. b) True. c) False. The pin icon toggles auto hide. d) True. e) False. The **Toolbox** contains the control icons. f) False. Some of a form's sizing handles are disabled. g) False. Forms have a title bar; labels do not. h) False. Control properties can be set using the **Properties** window. i) True. j) True.

EXERCISES

2.3 Fill in the blanks in each of the following statements:

a) The __________ button in the **Properties** window indicates that a dialog will appear.
b) To save every file in a solution, use the menu selection __________.
c) __________ help immediately brings up a relevant article. It can be accessed pressing the __________ key.
d) GUI stands for __________.

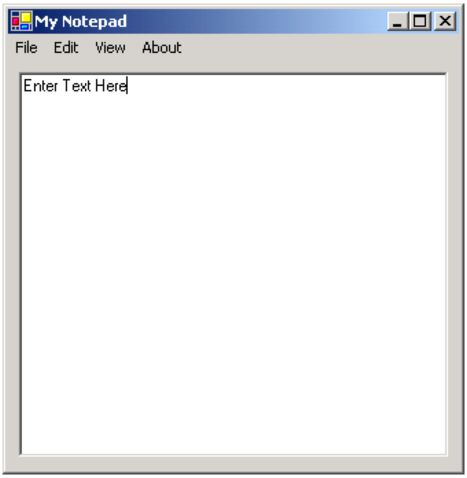
2.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Certain menu items appear only when designing a form.
b) The form, label and picture box have identical properties.
c) A person can browse the Internet from within Visual Studio .NET.
d) Visual C# programmers often create complex applications without writing any code.
e) Sizing handles are visible during execution.

2.5 Some features appear throughout Visual Studio .NET, performing similar actions in different contexts. Explain and give examples of how plus/minus icons, ellipsis buttons, down arrows and tool tips act in this manner. Why do you think Visual Studio .NET was designed to be this way?

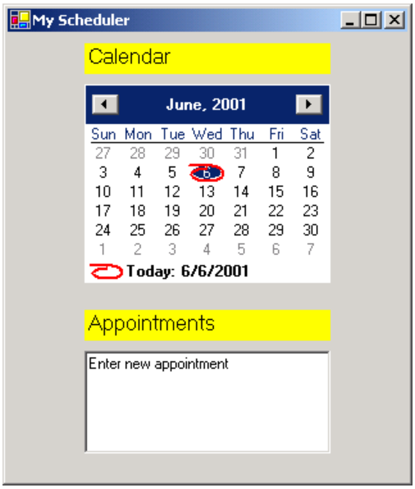
2.6 Build the GUIs described in each part of the exercise. (You need not provide any functionality.) Execute each program, and determine what happens when a control is clicked with the mouse. Drag controls from the **Toolbox** onto the form, and resize them as necessary.

a) This GUI consists of a **`MainMenu`** and a **`RichTextBox`**. Both controls can be dragged from the ToolBox onto the form or double clicked. After inserting the **`MainMenu`**, add items by clicking in the **Type Here** section, typing in the name of a menu item and pressing the *Enter* key. Resize the **`RichTextBox`** to fill the form.



Enter Text Here

b) This GUI consists of two `Label`s (font size 12, yellow background): a `MonthCalendar` and a `RichTextBox`. The calendar is displayed when the `MonthCalendar` is dragged on the form. The `MonthCalendar` and `RichTextBox` controls are similar to the controls we have seen previously. They can be dragged onto the form (or double clicked), then manipulated with the **Properties** window. [*Hint*: Use the `BackColor` property to change the background color of the labels.]



Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

Today: 6/6/2001

Enter new appointment

2.7 Fill in the blanks in each of the following statements:

a) The __________ property specifies which image a picture box displays.
b) The _________ has an icon in the **Toolbox**, but is not a control.
c) The _________ menu contains commands for arranging and displaying windows.
d) Property__________ determines a form's or control's background color.

2.8 Briefly describe each of the following IDE features:

a) toolbar
b) menu bar
c) toolbox
d) control
e) form
f) project
g) title bar

3

Introduction to C# Programming

Objectives

- To be able to write simple C# programs.
- To be able to use input and output statements.
- To become familiar with primitive data types.
- To understand basic memory concepts.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To be able to write decision-making statements.
- To be able to use relational and equality operators.

Comment is free, but facts are sacred.
C. P. Scott

The creditor hath a better memory than the debtor.
James Howell

When faced with a decision, I always ask, "What would be the most fun?"
Peggy Walker

Equality, in a social sense, may be divided into that of condition and that of rights.
James Fenimore Cooper

Outline

3.1 Introduction
3.2 Simple Program: Printing a Line of Text
3.3 Another Simple Program: Adding Integers
3.4 Memory Concepts
3.5 Arithmetic
3.6 Decision Making: Equality and Relational Operators

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

3.1 Introduction

This chapter introduces C# programming and presents examples that illustrate several important features of the language. Examples are analyzed one line at a time. In this chapter, we create *console applications*—applications that contain only text output. There are several types of projects that we can create in C#; the console application is one of the basic types. Text output in a console application is displayed in a *console window* (also called a *console window*). On Microsoft Windows 95/98, the console window is the ***MS-DOS prompt***. On Microsoft Windows NT/2000/XP, the console window is called the ***command prompt***. With C#, a program can be created with multiple types of output (windows, dialogs and so forth). These programs are called *Windows applications* and provide graphical user interfaces. We showed an example of a Windows application in Chapter 2, when we printed a message on a form. These types of applications will be discussed in greater detail, beginning with Chapter 4, Control Structures: Part 1 and Chapter 5, Control Structures: Part 2. In these chapters, we will also provide a detailed treatment of *program development* and *program control* in C#.

3.2 Simple Program: Printing a Line of Text

C# uses some notations that might appear strange to nonprogrammers. We begin by considering a simple program that displays a line of text. The program and its output are shown in Fig. 3.1. The program is followed by an output window that displays the program's results. When you execute this program, the output will appear in a console window.

```
// Fig. 3.1: Welcome1.cs
// A first program in C#.

using System;

class Welcome1
{
   static void Main( string[] args )
   {
      Console.WriteLine( "Welcome to C# Programming!" );
   }
}
```

Fig. 3.1 Our first program in C#. (Part 1 of 2.)

```
Welcome to C# Programming!
```

Fig. 3.1 Our first program in C#. (Part 2 of 2.)

This program illustrates several important features of C#. All programs we present in this book will include line numbers for the reader's convenience; these line numbers are not part of the C# programs. Line 10 in Fig. 3.1 does the "real work" of the program, displaying the phrase **`Welcome to C# Programming!`** on the screen.

Line 1 begins with `//`, indicating that the remainder of the line is a *comment*. Programmers insert comments to *document* and improve the readability of their code. Comments also help other people read and understand your programs. This comment simply indicates the figure number and file name for this program. We begin each program in this book in this manner. In this case, we have named the file **`Welcome1.cs`**. A comment that begins with `//` is called a *single-line comment*, because the comment terminates at the end of the line. Single-line comments can be placed almost anywhere in the program.

There is also a syntax for writing *multiple-line comments*. A multiple-line comment, such as

```
/* This is a multiple-line
   comment. It can be
   split over many lines */
```

begins with *delimiter* `/*` and ends with *delimiter* `*/`. All text between these delimiters is treated as a comment and is ignored by the compiler. In the Visual Studio .NET IDE, all comment text appears in green. Comments of the form `//` and `/*` … `*/` are ignored by the compiler; therefore, they do not cause the computer to perform any action when the program executes.

Common Programming Error 3.1

Forgetting one of the delimiters of a multiple-line comment is a syntax error. A syntax error *is caused when the compiler cannot recognize a statement. The compiler normally issues an error message to help the programmer locate and fix the incorrect statement. Syntax errors are violations of the language rules. Syntax errors are also called* compile errors, compile-time errors *or* compilation errors *because they are detected during the compilation phase. A program cannot compile or execute until all the syntax errors are corrected.*

Software Engineering Observation 3.1

Visual Studio will often times catch syntax errors as you are creating the program, even before the program is compiled. Look out for red jagged lines that may appear directly below a syntax error.

C# uses the same syntax as the C programming language for multiple-line comments (`/*`…`*/`) and the same syntax as C++ for single-line comments (`//`). C# programmers generally use C++-style single-line comments, instead of C-style comments. Throughout this book, we use mostly C++-style single-line comments.

Good Programming Practice 3.1

Every program should begin with one or more comments that describe the program's purpose.

Line 4 (known as a *`using` directive*) is generated by the Visual Studio IDE and declares that the program uses features in the *`System` namespace*. A *namespace* groups various C# features into related categories. One of the great strengths of C# is that C# programmers can use the rich set of namespaces provided by the .NET framework. These namespaces contain code that programmers can reuse, rather than "reinventing the wheel." This makes programming easier and faster. The namespaces that are defined in the .NET Framework contain preexisting code known as the *.NET Framework Class Library*. An example of one of the features in namespace `System` is `Console`, which we discuss momentarily. The various features are organized into namespaces that enable programmers to locate them easily. We discuss many namespaces and their features throughout the book.

Line 5 is a blank line. Programmers often use blank lines and space characters throughout a program to make the program easier to read. Collectively, blank lines, space characters, newline characters and tab characters are known as *whitespace* (space characters and tabs are known specifically as *whitespace characters*). *Newline characters* characters are "special characters" that indicate when to position the output cursor at the beginning of the next line in the console window to continue output. The compiler ignores blank lines, tabs and extra spaces that separate language elements. Several conventions for using whitespace characters are discussed in this and subsequent chapters.

Good Programming Practice 3.2

Use blank lines, space characters and tab characters in a program to enhance program readability.

Lines 6–12 define our first *class* (these lines collectively are called a *class definition*). C# programs consist of pieces called classes, which are logical groupings of members (e.g., *methods*) that simplify program organization. These methods (which are like functions in procedural programming languages) perform tasks and return information when the tasks are completed. A C# program consists of classes and methods created by the programmer and of preexisting classes found in the Framework Class Library. Throughout this book, we will teach the reader how to use both techniques in their programs. Every program in C# consists of at least one class definition that the programmer defines. These classes are known as *programmer-defined classes*. In Chapter 8, Object-Based Programming, we discuss programs that contain multiple programmer-defined classes. The *`class` keyword* begins a class definition in C# and is followed immediately by the *class name* (`Welcome1`, in this example). Keywords (or *reserved words*) are reserved for use by C# and always consist of lowercase letters. (A complete table of C# keywords is presented in the next chapter.) By convention, each word in a class name begins with an uppercase first letter and has an uppercase letter for each word in the class name (e.g., `SampleClassName`). The name of the class is known as an *identifier*, which is a series of characters consisting of letters, digits, underscores (_) and "at" symbols (`@`). Identifiers cannot begin with a digit and cannot contain spaces. Examples of valid identifiers are `Welcome1`, `_value`, `m_inputField1` and `button7`. The name `7button` is not a valid identifier because it begins with a digit, and the name `input field` is not a valid identifier because it contains a space. The "at" character (`@`) can be used only as the first character in an identifier. C# is *case sensitive*—uppercase and lowercase letters are considered different letters, so `a1` and `A1` are different identifiers.

Common Programming Error 3.2

C# is case sensitive. Not using the proper case for an identifier, e.g.,writing **`Total`** *when the identifier is* **`total`***, is a compiler error.*

Good Programming Practice 3.3

Always begin a class name with an uppercase first letter. This practice makes class names easier to identify.

The *left brace* (**`{`**) at line 7 begins the *body of the class definition*. The corresponding *right brace* (**`}`**) at line 12 ends the class definition. Notice that lines 8–11 in the body of the class are indented. This is one of the spacing conventions mentioned earlier. Indentation improves program readability. We define each spacing convention as a *Good Programming Practice*.

Common Programming Error 3.3

If braces do not occur in matching pairs, a syntax error occurs.

Good Programming Practice 3.4

When typing an opening left brace (`{`*) in a program, immediately type the closing right brace (*`}`*) then reposition the cursor between the braces to begin typing the body. This practice helps prevent missing braces. Readers may notice that, when they type the closing brace, Visual Studio .NET makes both braces bold (as well as the first line of the class definition). This is helpful in the creation of more complex programs that involve multiple sets of opening and closing braces.*

Good Programming Practice 3.5

Indent the entire body of each class definition one "level" of indentation between the left brace (`{`*) and the right brace (*`}`*) that delimit the class body. This emphasizes the structure of the class definition and helps make the class definition easier to read. Visual Studio .NET provides indentation in several places as programmers enter code.*

Line 8 is present in all C# console and Windows applications. These applications begin executing at ***`Main`***, which is known as the *entry point* of the program. The parentheses after **`Main`** indicate that **`Main`** is a program building block, called a method. C# class definitions normally contain one or more methods and C# applications contain one or more classes. For a C# console or Windows application, exactly one of those methods must be called **`Main`**, and it must be defined as shown on line 8; otherwise, the program is not executable. Normally, a console applications's **`Main`** method is defined as shown on line 8. Methods are explained in detail in Chapter 6, Methods. For now, simply mimic **`Main`**'s first line in each C# application.

The left brace (**`{`**) on line 9 begins the *body of the method definition* (the code which will be executed as a part of our program). A corresponding right brace (**`}`**) terminates the method definition's body (line 11). Notice that the line in the body of the method is indented between these braces.

Good Programming Practice 3.6

Indent the entire body of each method definition one "level" of indentation between the left brace (`{`*) and the right brace (*`}`*) that define the method body. This makes the structure of the method stand out, improving the method definition's readability.*

Line 10 instructs the computer to perform an *action*, namely, to print a series of characters contained between the double quotation marks. Characters delimited in this manner are called *strings*, *character strings* or *string literals*. We refer to characters between double quotation marks generically as strings. Whitespace characters in strings are significant—the compiler does not ignore these characters when they appear in strings.

The *`Console`* *class* enables programs to output information to the computer's *standard output*. Class `Console` provides methods that allow C# programs to display strings and other types of information in the Windows command prompt.

Method *`Console.WriteLine`* *displays* (or *prints*) a line of text in the console window. When `Console.WriteLine` completes its task, it positions the *output cursor* (the location where the next character will be displayed) at the beginning of the next line in the console window. (This is similar to pressing the *Enter* key when typing in a text editor—the cursor is repositioned at the beginning of the next line in the file.)

The entire line, including `Console.WriteLine`, its *argument* in the parentheses (`"Welcome to C# Programming!"`) and the *semicolon* (`;`), is called a *statement*. Every statement must end with a semicolon (known as the *statement terminator*). When this statement executes, it displays the message `Welcome to C# Programming!` in the console window (Fig. 3.1).

In C# statements we normally precede each class name with its namespace name and a period. For example, line 10 would normally be

```
System.Console.WriteLine( "Welcome to C# Programming!" );
```

for the program to run correctly. The `using` directive on line 4 eliminates the need to specify explicitly the namespace `System` when using classes in the namespace. This can save time and confusion for programmers.

Common Programming Error 3.4

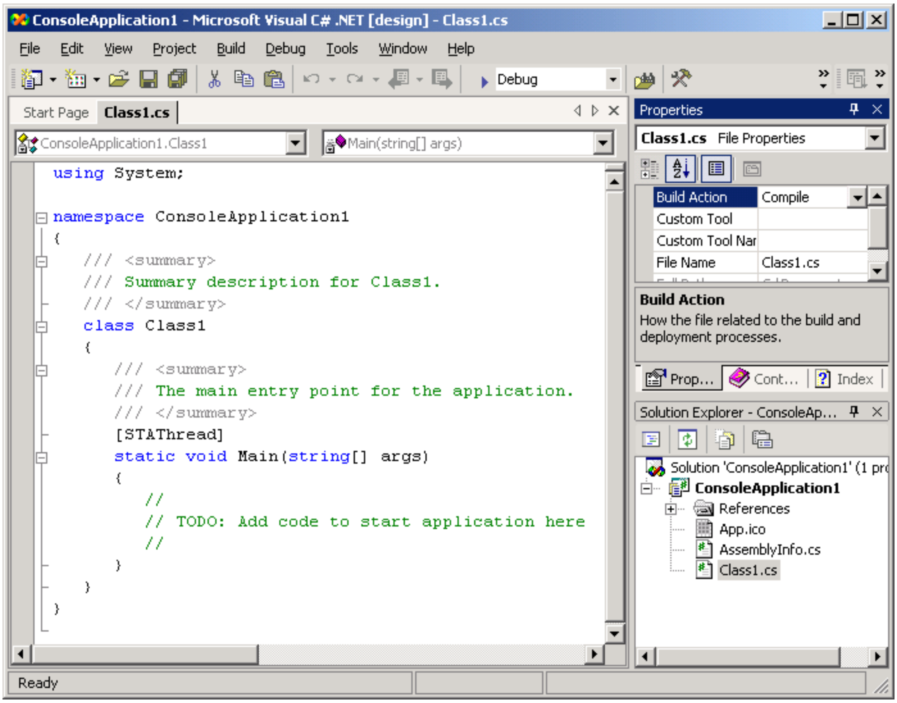
Omitting the semicolon at the end of a statement is a syntax error.

Testing and Debugging Tip 3.1

When the compiler reports a syntax error, the error might not be on the line indicated by the error message. First, check the line where the error was reported. If that line does not contain syntax errors, check the lines that precede the one reported.

Now that we have presented this program to you, let us explain step-by-step how to create and run it in Visual Studio.

1. *Create the console application.* Go to the **File** menu and choose **New**, then **Project…**. A dialog will appear. In the left pane, choose **Visual C# Projects**; from the right pane, choose ***Console Application***. It is possible to specify other information about the project in the bottom portion of this dialog (i.e., the name and location of the project). After entering all the necessary information, click **OK** to create the project. The project is created, and the code window is opened for editing. The new application is shown in Fig. 3.2. Note that this is the same way we created our application in Chapter 2, except that now we have chosen a console application, instead of a Windows application.



ConsoleApplication1 - Microsoft Visual C# .NET [design] - Class1.cs
File Edit View Project Build Debug Tools Window Help
Debug
Start Page
Class1.cs
ConsoleApplication1.Class1
Main(string[] args)
Properties
Class1.cs File Properties
Prop...
Cont...
Index
Solution Explorer - ConsoleAp...
Solution 'ConsoleApplication1' (1 pro
ConsoleApplication1
References
App.ico
AssemblyInfo.cs
Class1.cs
Ready

```
using System;

namespace ConsoleApplication1
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   class Class1
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
      {
         //
         // TODO: Add code to start application here
         //
      }
   }
}
```

Build Action	Compile
Custom Tool	
Custom Tool Nar	
File Name	Class1.cs

Build Action
How the file related to the build and deployment processes.

Fig. 3.2 Visual Studio .NET-generated console application.

This application can be built (compiled) and executed, but will not do anything until we add more code (this is done in Step 3). Let us briefly look at the code generated for us by the IDE.

Notice that this code contains features that we have not yet discussed. We have done this for both display and clarity reasons—at this point in the book, this code is neither required nor relevant to the discussion of this program. Much of the extra code that the IDE provides is used either for documentation or to help create graphical user interfaces. One of the things that the reader will no doubt notice is that we do not show the lines directly above and below the class definition. These lines are used to create namespaces, a topic that will be discussed in Chapter 8, Object-Based Programming. [*Note*: Several times early in this text, we ask the reader to mimic certain C# features that we introduce. We do this especially when it is not yet important to know all the details of a feature to use that feature in C#. All programmers initially learn how to program by mimicking what other programmers have done. For each detail, we ask the reader to mimic, we indicate where the full discussion will be presented later in the text.] The code for all examples in the book is included for the reader on our Web site `www.deitel.com` under the **Downloads/Resources** link.

2. *Change the name of the program file.* For the programs in this book, we usually change the name of the code file. By default, the file is named `Class1.cs`. This can be changed by right-clicking the name of the file in the **Solution Explorer**

and selecting **Rename**. The reader can then enter a new name for the file, provided that this file ends in `.cs` (the file extension for C# code files).

3. *Complete the code.* In the text editor, replace the comment

```
//
// TODO: Add code to start application here
//
```

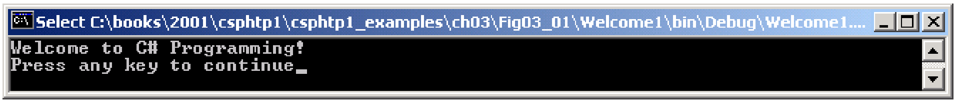
which is located within method `Main` with line 10 from Fig. 3.1 (this comment is no longer necessary, for we are adding code to the program).

4. *Run the program.* We are now ready to compile and execute our program. To do this, we simply follow the same steps that we executed for the example in Chapter 2. To compile the program, go to the **Build** menu and select **Build Solution**. If the program contains no syntax errors, the preceding command creates a new file called `Welcome1.exe`, containing the MSIL code for our application. To execute this program, choose option **Start Without Debugging**[1] in the **Debug** menu.

Program execution begins with method `Main`, which is the entry point to the program. Next, the statement at line 10 of `Main` displays `Welcome to C# Programming!` Figure 3.3 shows result of executing the program.

The message `Welcome to C# Programming!` can be displayed via multiple method calls. Class `Welcome2` of Fig. 3.4 uses two statements to produce the same output shown in Fig. 3.3.

Lines 10–11 of Fig. 3.4 display one line in the console window. The first statement calls `Console` method `Write` to display a string. Unlike `WriteLine`, `Write` does not position the output cursor at the beginning of the next line in the console window after displaying its string. The next character displayed in the console window appears immediately after the last character displayed with `Write`. Thus, when line 11 executes, the first character displayed (`C`) appears immediately after the last character displayed with `Write` (i.e., the space character after the word `"to"` in line 10). Each `Write` or `WriteLine` statement resumes displaying characters from where the last `Write` or `WriteLine` stopped.



Select C:\books\2001\csphtp1\csphtp1_examples\ch03\Fig03_01\Welcome1\bin\Debug\Welcome1....

```
Welcome to C# Programming!
Press any key to continue_
```

Fig. 3.3 Execution of the `Welcome1` program.

---

[1] Selecting **Debug > Start Without Debugging** causes the command window to prompt the user to press a key after the program terminates, allowing the user to observe the program's output. In contrast, if we run this program using **Debug > Start**, as we did for the Windows application in Chapter 2, a command window opens, the program displays the message **Welcome to C# Programming!**, then the command window closes immediately.

```
1   // Fig. 3.4: Welcome2.cs
2   // Printing a line with multiple statements.
3
4   using System;
5
6   class Welcome2
7   {
8      static void Main( string[] args )
9      {
10        Console.Write( "Welcome to " );
11        Console.WriteLine( "C# Programming!" );
12     }
13  }
```

```
Welcome to C# Programming!
```

Fig. 3.4 Printing on one line with separate statements.

A single statement can display multiple lines by using newline characters. Recall that these characters indicate when to position the output cursor at the beginning of the next line in the console window to continue output. Figure 3.5 demonstrates using newline characters.

Line 10 produces four separate lines of text in the console window. Normally, the characters in a string are displayed exactly as they appear between the double quotes. However, notice that the two characters “****” and “**n**” do not appear on the screen. The *backslash* (****) is called an *escape character*. It indicates that a “special” character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an *escape sequence*. This escape sequence **\n** is the *newline character*. It causes the *cursor* (i.e., the current screen position indicator) to move to the beginning of the next line in the console window. Some common escape sequences are listed in Fig. 3.6.

```
1   // Fig. 3.5: Welcome3.cs
2   // Printing multiple lines with a single statement.
3
4   using System;
5
6   class Welcome3
7   {
8      static void Main( string[] args )
9      {
10        Console.WriteLine( "Welcome\nto\nC#\nProgramming!" );
11     }
12  }
```

```
Welcome
to
C#
Programming!
```

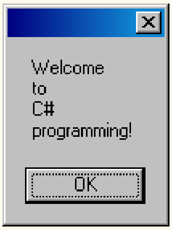
Fig. 3.5 Printing on multiple lines with a single statement.

Escape sequence	Description
`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the previous characters output on that line.
`\\`	Backslash. Used to print a backslash character.
`\"`	Double quote. Used to print a double quote (`"`) character.

Fig. 3.6 Some common escape sequences.

Although the first several programs display output in the command prompt, most C# applications use windows or *dialogs* to display output. As mentioned earlier, dialogs are windows that typically display important messages to the user of an application. The .NET Framework Class Library includes class ***`MessageBox`*** for creating dialogs. Class **`MessageBox`** is defined in namespace ***`System.Windows.Forms`***. The program in Fig. 3.7 displays the same string as Fig. 3.5 in a message dialog using class **`MessageBox`**.

```
// Fig. 3.7: Welcome4.cs
// Printing multiple lines in a dialog Box.

using System;
using System.Windows.Forms;

class Welcome4
{
   static void Main( string[] args )
   {
      MessageBox.Show( "Welcome\nto\nC#\nprogramming!" );
   }
}
```



Welcome
to
C#
programming!

Fig. 3.7 Displaying multiple lines in a dialog.

Many compiled classes in C# (including `MessageBox`) need to be referenced before they can be used in a program. Depending on the type of application we create, classes may be compiled into files with a *`.exe`* (*executable*) extension, a *`.dll`* (or *dynamic link library*) extension or one of several other extensions. Such files are called *assemblies* and are the packaging units for code in C#. [Note: Assemblies can be comprised of many files of several different types.] Namespaces group related *classes* together; the assembly is a package containing the Microsoft Intermediate Language (MSIL) code that a project has been compiled into, plus any other information that is needed for these classes. The assembly that we need to reference can be found in the Visual Studio .NET documentation (also called the MSDN Documentation) for the class we wish to use. The easiest way to access this information is to go to the **Help** menu in Visual Studio, and choose **Index**. The reader can then type in the name of the class to access the documentation. Class `MessageBox` is located in assembly `System.Windows.Forms.dll`. As mentioned previously, we must add a reference to this assembly to use class `MessageBox` in our program. Let us discuss an example of adding a reference to `System.Windows.Forms` within the IDE.

Common Programming Error 3.5

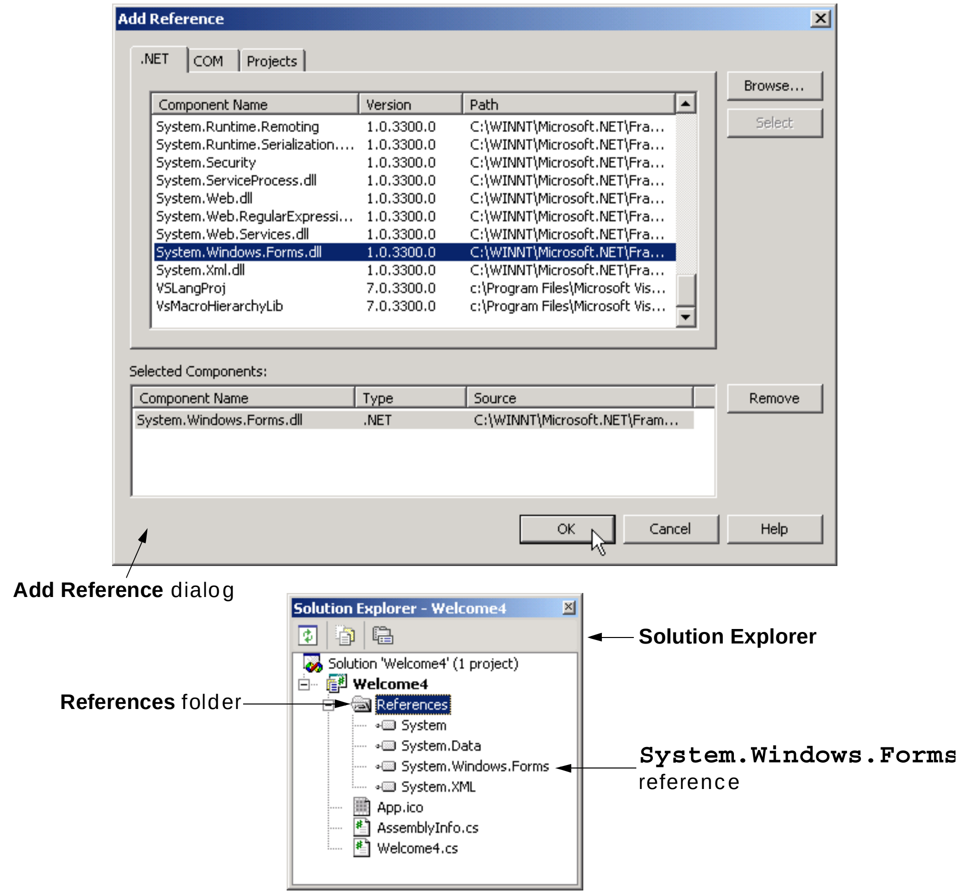
Including a namespace with the `using` directive, but not adding a reference to the proper assembly, results in a compiler error.

To begin, make sure you have an application open. Select the ***Add Reference…*** option from the **Project** menu, or right click the **References** folder in the **Solution Explorer** and select **Add Reference…** from the popup menu that appears. This opens the **Add Reference** dialog (Fig. 3.8). Double click `System.Windows.Forms.dll` to add this file to the **Selected Components** list at the bottom of the dialog, then click **OK**. Notice that `System.Windows.Forms` now appears in the **References** folder of the **Solution Explorer** (Fig. 3.8).

After referencing the appropriate assembly and providing a `using` directive for the corresponding namespace (line 5), we can use the classes in that namespace (such as `MessageBox`).

The reader may notice that we did not add any references to our previous programs. Visual Studio adds a few common references when a project is created. Also, by default, some assemblies do not require references. Class `Console`, for instance, is located in the assembly `mscorlib.dll`, but a reference to this assembly is not required to use it.

The `System.Windows.Forms` namespace contains many classes that help C# programmers define graphical user interfaces (GUIs) for their applications. *GUI components* (e.g., buttons) facilitate both data entry by the user and the formatting or presenting of data outputs to the user. For example, Fig. 3.9 is an Internet Explorer window with a bar containing menus (**File**, **Edit**, **View** etc.). Below the menu bar there is a set of buttons, each with a defined task in Internet Explorer. Below the buttons there is a *text field* in which the user can type the location of a World Wide Web site to visit. To the left of the text field is a *label* that indicates the purpose of the text field. The menus, buttons, text fields and labels are part of Internet Explorer's GUI. They enable users to interact with the Internet Explorer program. C# contains classes that create the GUI components described here. Other classes that create GUI components will be described in Chapters 12 and 13, Graphical User Interfaces: Part 1 and Graphical User Interfaces: Part 2.



Add Reference

Component Name	Version	Path
System.Runtime.Remoting	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Runtime.Serialization....	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Security	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.ServiceProcess.dll	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Web.dll	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Web.RegularExpressi...	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Web.Services.dll	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Windows.Forms.dll	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
System.Xml.dll	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
VSLangProj	7.0.3300.0	c:\Program Files\Microsoft Vis...
VsMacroHierarchyLib	7.0.3300.0	c:\Program Files\Microsoft Vis...

Selected Components:

Component Name	Type	Source
System.Windows.Forms.dll	.NET	C:\WINNT\Microsoft.NET\Fram...

.NET
COM
Projects
Browse...
Select
Remove
OK
Cancel
Help
Add Reference dialog
Solution Explorer - Welcome4
Solution 'Welcome4' (1 project)
Welcome4
References
System
System.Data
System.Windows.Forms
System.XML
App.ico
AssemblyInfo.cs
Welcome4.cs
Solution Explorer
References folder
System.Windows.Forms reference

Fig. 3.8 Adding a reference to an assembly in Visual Studio .NET.

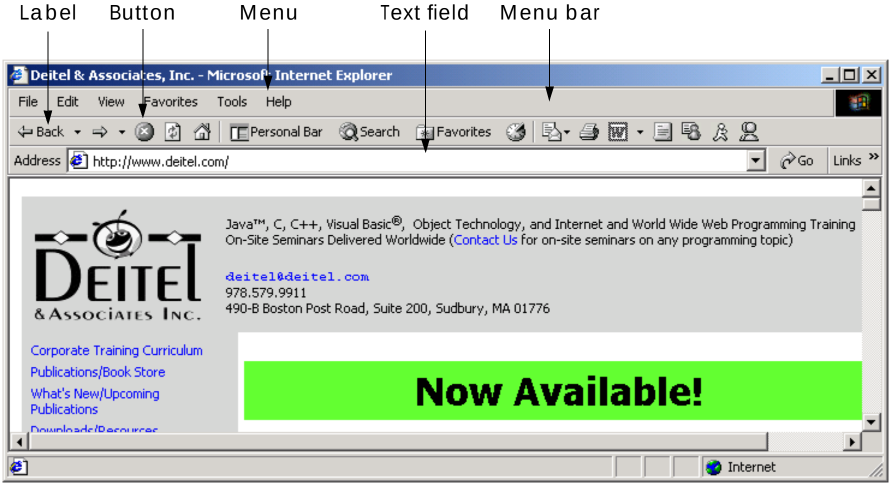
In **`Main`**, line 11 calls method ***`Show`*** of class **`MessageBox`** (Fig. 3.7). This method takes a string as an argument and displays it to the user in a message dialog. Method **`Show`** is called a *static method*. Such methods are always called by using their class name (in this case, **`MessageBox`**) followed by a *dot operator* (**`.`**) and the method name (in this case, **`Show`**). We discuss static methods in Chapter 8, Object-Based Programming.

Line 11 displays the dialog box shown in Fig. 3.10. The dialog includes an **OK** button that allows the user to *dismiss (close)* the dialog. Positioning the *mouse cursor* (also called the *mouse pointer*) over the **OK** button and clicking the mouse dismisses the dialog.

C# allows large statements to be split over many lines. For example, we could have split the statement on line 11 into the following two lines:

```
MessageBox.Show(
   "Welcome\nto\nC#\nprogramming!" );
```

All statements end with a semicolon (**`;`**), so the compiler recognizes that these two lines represent only one statement. However, you cannot split a statement in the middle of an identifier (e.g., the class name) or a string.


Label
Button
Menu
Text field
Menu bar
Deitel & Associates, Inc. - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Personal Bar
Search
Favorites
Address
http://www.deitel.com/
Go
Links
Internet

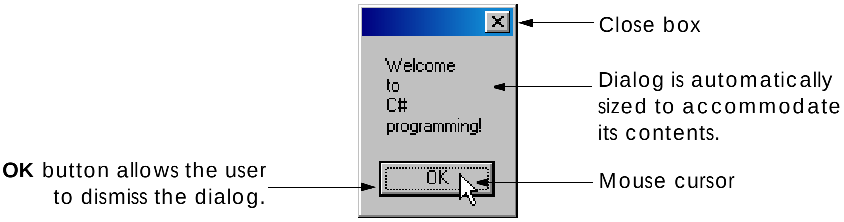
Java™, C, C++, Visual Basic®, Object Technology, and Internet and World Wide Web Programming Training
On-Site Seminars Delivered Worldwide (Contact Us for on-site seminars on any programming topic)

deitel@deitel.com
978.579.9911
490-B Boston Post Road, Suite 200, Sudbury, MA 01776

Corporate Training Curriculum
Publications/Book Store
What's New/Upcoming Publications

Now Available!

Fig. 3.9 Internet Explorer's GUI.


Close box
Dialog is automatically sized to accommodate its contents.
Mouse cursor
OK button allows the user to dismiss the dialog.
OK

Welcome
to
C#
programming!

Fig. 3.10 Dialog displayed by calling `MessageBox.Show`.

Common Programming Error 3.6

Splitting a statement in the middle of an identifier or a string is a syntax error.

The user can close the dialog by clicking the **OK** button or the close box. Once this occurs, the program terminates, because the `Main` method terminates.

3.3 Another Simple Program: Adding Integers

Our next application (Fig. 3.11) inputs two integers (whole numbers) typed by a user at the keyboard, computes the sum of these values and displays the result. As the user types each integer and presses the *Enter* key, the integer is read into the program and added to the total. Lines 1–2 are single-line comments stating the figure number, file name and purpose of the program.

As stated previously, every C# program consists of at least one class definition. Line 6 begins the definition of class `Addition`. Lines 7–37 define the body of the class. Recall that all class definitions start with an opening left brace (`{`) and end with a closing right brace (`}`).

```
 1  // Fig. 3.11: Addition.cs
 2  // An addition program.
 3
 4  using System;
 5
 6  class Addition
 7  {
 8     static void Main( string[] args )
 9     {
10        string firstNumber,   // first string entered by user
11               secondNumber;  // second string entered by user
12
13        int number1,          // first number to add
14            number2,          // second number to add
15            sum;              // sum of number1 and number2
16
17        // prompt for and read first number from user as string
18        Console.Write( "Please enter the first integer: " );
19        firstNumber = Console.ReadLine();
20
21        // read second number from user as string
22        Console.Write( "\nPlease enter the second integer: " );
23        secondNumber = Console.ReadLine();
24
25        // convert numbers from type string to type int
26        number1 = Int32.Parse( firstNumber );
27        number2 = Int32.Parse( secondNumber );
28
29        // add numbers
30        sum = number1 + number2;
31
32        // display results
33        Console.WriteLine( "\nThe sum is {0}.", sum );
34
35     } // end method Main
36
37  } // end class Addition
```

```
Please enter the first integer: 45

Please enter the second integer: 72

The sum is 117.
```

Fig. 3.11 Addition program that adds two values entered by the user.

The program begins execution with method **`Main`** on line 8. The left brace (line 9) begins **`Main`**’s body and the corresponding right brace (line 35) terminates **`Main`**’s body.

Lines 10–11 are a *declaration*. The words **`firstNumber`** and **`secondNumber`** are the names of *variables*. A variable is a location in the computer’s memory where a value can be stored for use by a program. All variables must be declared with a name and a data type before they can be used in a program. This declaration specifies that the variables **`firstNumber`** and **`secondNumber`** are data of type ***`string`***, which means that these variables store strings of characters. There are certain data types already defined in the .NET Framework, known as *built-in data types* or *primitive data types*. Types such as `string`, `int`, `double` and `char` are examples of primitive data types. Primitive type names are keywords. The 15 primitive types are summarized in Chapter 4, Control Structures: Part 1.

A variable name can be any valid identifier. Declarations end with a semicolon (`;`) and can be split over several lines with each variable in the declaration separated by a comma (i.e., a *comma-separated list* of variable names). Several variables of the same type may be declared in one or in multiple declarations. We could have written two declarations, one for each variable, but the preceding declaration is more concise. Notice the single-line comments at the end of each line. This is a common syntax used by programmers to indicate the purpose of each variable in the program.

Good Programming Practice 3.7

Choosing meaningful variable names helps a program to be "self-documenting" (i.e., easier to understand simply by reading it, rather than having to read manuals or use excessive comments).

Good Programming Practice 3.8

By convention, variable-name identifiers begin with a lowercase letter. As with class names, every word in the name after the first word should begin with a capital letter. For example, identifier `firstNumber` has a capital `N` in its second word, `Number`.

Good Programming Practice 3.9

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a comment that describes each variable.

Lines 13–15 declare that variables `number1`, `number2` and `sum` are of data type `int`, which means that these variables will hold *integer* values (i.e., whole numbers such as –11, 7, 0 and 31914). In contrast, the data types `float` and `double` specify real numbers (i.e., floating-point numbers with decimal points, such as 3.4, 0.0 and –11.19) and variables of type `char` specify character data. A `char` variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single character, such as `x`, `$`, `7`, `*` and escape sequences (like as the newline character `\n`). Oftentimes in programs, characters are denoted in single quotes, such as `'x'`, `'$'`, `'7'`, `'*'` and `'\n'`, to differentiate between a value and a variable name. C# is also capable of representing all Unicode characters. *Unicode* is an extensive international *character set* (collection of characters) that enables the programmer to display letters in different languages, mathematical symbols and much more. For more information on this topic, see Appendix G, Unicode.

Lines 18–19 prompt the user to input an integer and read from the user a `string` representing the first of the two integers that the program will add. The message on line 18 is called a *prompt*, because it directs the user to take a specific action. Method `ReadLine` (line 19) causes the program to pause and wait for user input. The user inputs characters from the keyboard, then presses the *Enter* key to return the string to the program. Unfortunately, the .NET Framework does not provide a simple input dialog. For this reason, the examples in these early chapters receive user input through the command prompt.

Technically, the user can send anything to the program as input. For this program, if the user types a noninteger value, a *run-time logic error* (an error that has its effect at execution time) occurs. Chapter 11, Exception Handling, discusses how to make your programs more robust by handling such errors.

When the user enters a number and presses *Enter*, the program assigns the string representation of this number to variable **`firstNumber`** (line 19) with the *assignment operator* **`=`**. The statement is read as, "**`firstNumber`** *gets* the value returned by method **`ReadLine`**." The **`=`** operator is a *binary operator*, because it has two *operands*—**`firstNumber`**, and the result of the expression **`Console.ReadLine`**. The entire statement is an *assignment statement*, because it assigns a value to a variable. In an assignment statement, first the right side of the assignment is evaluated, then the result is assigned to the variable on the left side of the assignment. So, line 19 executes method **`ReadLine`**, then assigns the string value to **`firstNumber`**.

Good Programming Practice 3.10

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

Lines 22–23 prompt the user to enter a second integer and read from the user a string representing the value. Lines 26–27 convert the two strings input by the user to **`int`** values that can be used in a calculation. Method ***`Int32.Parse`*** (a static method of class **`Int32`**) converts its **`string`** argument to an integer. Class **`Int32`** is part of the **`System`** namespace. Line 26 assigns the integer that **`Int32.Parse`** returns to variable **`number1`**. Any subsequent references to **`number1`** in the program use this integer value. Line 27 assigns the integer that **`Int32.Parse`** returns to variable **`number2`**. Any subsequent references to **`number2`** in the program use this integer value. You can eliminate the need for **`string`** variables **`firstNumber`** and **`secondNumber`** by combining the input and conversion operations as follows:

```
int number1;
number1 = Int32.Parse( Console.ReadLine() );
```

In C#, users input data as **`string`**s. We convert these strings to perform integer arithmetic. Arithmetic operations, as we will discuss in Section 3.5, do not work with **`string`**s the same way operations work with integers. To add numbers and get the proper sum, we must convert the strings to integers. The preceding statements do not make use of the **`string`** variable (**`firstNumber`**). This variable is required only to store the **`string`** temporarily until the program converts it. Reading the **`string`** and converting it on one line makes the variable unnecessary.

The assignment statement on line 30 calculates the sum of the variables **`number1`** and **`number2`** and assigns the result to variable **`sum`** by using the assignment operator **`=`**. The statement is read as, "**`sum`** *gets* the value of **`number1`** plus **`number2`**." Most calculations are performed in assignment statements.

After performing the calculation, line 33 displays the result of the addition. In this example, we want to output the value in a variable using method **`WriteLine`**. Let us discuss how this is done.

The *comma-separated* arguments to **`Console.WriteLine`**

```
"\nThe sum is {0}.", sum
```

use **`{0}`** to indicate a placeholder for a variable's value. If we assume that **`sum`** contains the value **`117`**, the expression evaluates as follows: Method **`WriteLine`** encounters a number in curly braces, `{0}`, known as a *format*. This indicates that the variable found after the string in the list of arguments (in this case, `sum`) will be evaluated and incorporated into our string, in place of the format. The resulting string will be “`The sum is 117`.” Similarly, in the statement

```
Console.WriteLine(
   "The numbers entered are {0} and {1}", number1, number2 );
```

the value of `number1` would replace `{0}` (because it is the first variable) and the value of `number2` would replace `{1}` (because it is the second variable). The resulting string would be `"The numbers entered are 45 and 72"`. More formats can be used (`{2}`, `{3}` etc.) if there are more variables to display in the string.

Good Programming Practice 3.11

Place a space after each comma in a method’s argument list to make programs more readable.

Some programmers find it difficult, when reading or writing a program, to match the left and right braces (`{` and `}`) that delimit the body of a class or method definition. For this reason, some programmers include a single-line comment after each closing right brace that ends a method or class definition, as we do in lines 35 and 37.

Good Programming Practice 3.12

Follow the closing right brace (`}`) of the body of a method or class definition with a single-line comment. This comment should indicate the method or class that the right brace terminates.

3.4 Memory Concepts

Variable names, such as `number1`, `number2` and `sum`, actually correspond to *locations* in the computer’s memory. Every variable has a *name*, a *type*, a *size* and a *value.*

In the addition program in Fig. 3.11, the statement (line 26)

```
number1 = Int32.Parse( firstNumber );
```

converts to an `int` the string that the user entered. This `int` is placed into a memory location to which the name `number1` has been assigned by the compiler. Suppose the user enters the string `45` as the value for `firstNumber`. The program converts `firstNumber` to an `int`, and the computer places the integer value `45` into location `number1`, as shown in Fig. 3.12.

When a value is placed in a memory location, this value replaces the previous value in that location. The previous value is lost (or destroyed).

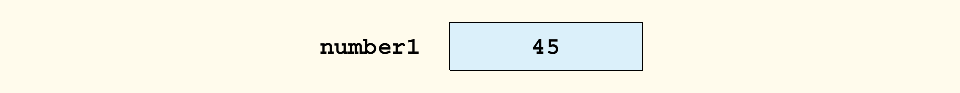


Fig. 3.12 Memory location showing name and value of variable `number1`.

When the statement (line 27)

```
number2 = Int32.Parse( secondNumber );
```

executes, suppose the user types `72` as the value for `secondNumber`. The program converts `secondNumber` to an `int`, the computer places the integer value `72` into location `number2` and memory appears as shown in Fig. 3.13.

Once the program has obtained values for `number1` and `number2`, it adds these values and places their total into variable `sum`. The statement

```
sum = number1 + number2;
```

performs the addition and replaces (i.e., destroys) `sum`'s previous value. After calculating the `sum`, memory appears as shown in Fig. 3.14. Note that the values of `number1` and `number2` appear exactly as they did before the calculation of `sum`. These values were used, but not destroyed, as the computer performed the calculation. Thus, when a value is read from a memory location, the process is *nondestructive*.

3.5 Arithmetic

Most programs perform arithmetic calculations. Figure 3.15 summarizes the *arithmetic operators*. Note the use of various special symbols not used in algebra. The *asterisk* (`*`) indicates multiplication, and the *percent sign* (`%`) represents the *modulus operator*, which is discussed shortly. The arithmetic operators in Fig. 3.15 are binary operators, because they each require two operands. For example, the expression `sum + value` contains the binary operator `+` and the two operands `sum` and `value`.

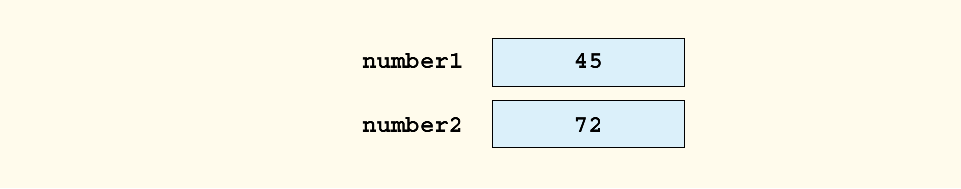


Fig. 3.13 Memory locations after values for variables `number1` and `number2` have been input.

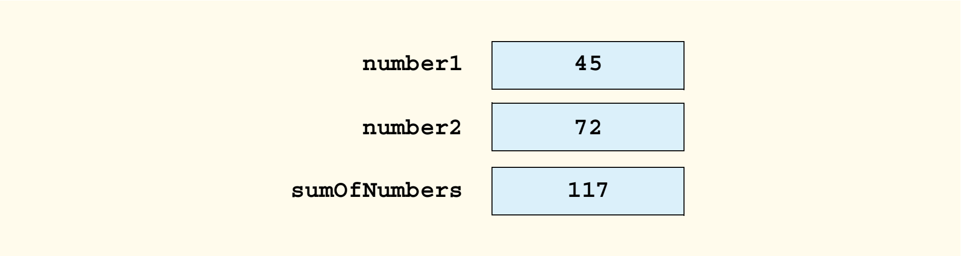


Fig. 3.14 Memory locations after a calculation.

C# operation	Arithmetic operator	Algebraic expression	C# expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	`%`	$r \bmod s$	`r % s`

Fig. 3.15 Arithmetic operators.

Integer division contains two `int` operands. The result of this computation is an integer quotient; for example, the expression `7 / 4` evaluates to `1` and the expression `17 / 5` evaluates to `3`. Note that any fractional part in integer division simply is discarded (i.e., truncated)—no rounding occurs. C# provides the modulus operator, `%`, which yields the remainder after integer division. The expression `x % y` yields the remainder after `x` is divided by `y`. Thus, `7 % 4` yields `3` and `17 % 5` yields `2`. This operator is used most commonly with integer operands, but also can be used with other arithmetic types. In later chapters, we consider interesting applications of the modulus operator, such as determining whether one number is a multiple of another. There is no arithmetic operator for exponentiation in C#. (Chapter 6, Methods, discusses how to perform exponentiation in C#.)

Arithmetic expressions in C# must be written in *straight-line form* to facilitate entering programs into a computer. Thus, expressions such as "`a` divided by `b`" must be written as `a / b` so that all constants, variables and operators appear in a straight line. The following algebraic notation generally is not acceptable to compilers:

$$\frac{a}{b}$$

C# expressions can use parentheses in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c`, we write

```
a * ( b + c )
```

C# applies the operators in arithmetic expressions in a precise sequence, determined by the following *rules of operator precedence*, which are generally the same as those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Thus, *parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer*. Parentheses are at the highest level of precedence. With *nested* (or *embedded*) parentheses, the operators in the innermost pair of parentheses are applied first.

2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are

applied from left to right. Multiplication, division and modulus are said to have the same level of precedence.

3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from left to right. Addition and subtraction have the same level of precedence.

The rules of operator precedence enable C# to apply operators in the correct order. When we say operators are applied from left to right, we are referring to the *associativity* of the operators. If there are multiple operators, each with the same precedence, the associativity determines the order in which the operators are applied. We will see that some operators associate from right to left. Figure 3.16 summarizes the rules of operator precedence. This table will expand as we introduce additional C# operators in subsequent chapters. See Appendix A for a complete operator-precedence chart.

Notice in the chart that we make note of nested parentheses. Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

```
a * ( b + c ) + c * ( d + e )
```

has multiple sets of parentheses, but not nested parentheses. Rather, these parentheses are said to be “on the same level.”

Let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C# equivalent.

The following is an example of an arithmetic mean (average) of five terms:

$$\text{Algebra:} \quad = \frac{a + b + c + d + e}{5}$$

C#: `m = ( a + b + c + d + e ) / 5;`

The parentheses are required because division has higher precedence than addition. The entire quantity `( a + b + c + d + e )` is to be divided by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5`, which evaluates as

Operator(s)	Operation	Order of evaluation (precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right.
`*`, `/` or `%`	Multiplication Division Modulus	Evaluated second. If there are several such operators, they are evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several such operators, they are evaluated left to right.

Fig. 3.16 Precedence of arithmetic operators.

$$+ b + c + d + \frac{e}{5}$$

The following is the equation of a straight line:

Algebra: $y = mx + b$

C#:
```
y = m * x + b;
```

No parentheses are required. The multiplication occurs first because multiplication has a higher precedence than addition. The assignment occurs last because it has a lower precedence than multiplication and division.

The following example contains modulus (`%`), multiplication, division, addition and subtraction operations:

Algebra: $z = pr\%q + w/x - y$

C#:
```
z  =  p  *  r  %  q  +  w  /  x  -  y;
```

⑥ ① ② ④ ③ ⑤

The circled numbers under the statement indicate the order in which C# applies the operators. The multiplication, modulus and division operators evaluate first in left-to-right order (i.e., they associate from left to right). The addition and subtraction evaluate next. These also are applied from left to right.

To develop a better understanding of the rules of operator precedence, consider how a second-degree polynomial ($y = ax^2 + bx + c$) evaluates:

```
y  =  a  *  x  *  x  +  b  *  x  +  c;
```

⑥ ① ② ④ ③ ⑤

The circled numbers under the statement indicate the order in which C# applies the operators. There is no arithmetic operator for exponentiation in C#; x^2 is represented as **`x * x`**. The .NET Framework Class Library provides method **`Math.Pow`** for exponentiation (see Chapter 6, Methods).

Suppose **`a`**, **`b`**, **`c`** and **`x`** are initialized as follows: **`a = 2`**, **`b = 3`**, **`c = 7`** and **`x = 5`**. Figure 3.17 illustrates the order of evaluation of the operators.

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression easier to read. Unnecessary parentheses are also called *redundant parentheses*. For example, the preceding assignment statement might be parenthesized as

```
y = ( a * x * x ) + ( b * x ) + c;
```

Good Programming Practice 3.13

Using parentheses for more complex arithmetic expressions, even when the parentheses are not necessary can make the arithmetic expressions easier to read.

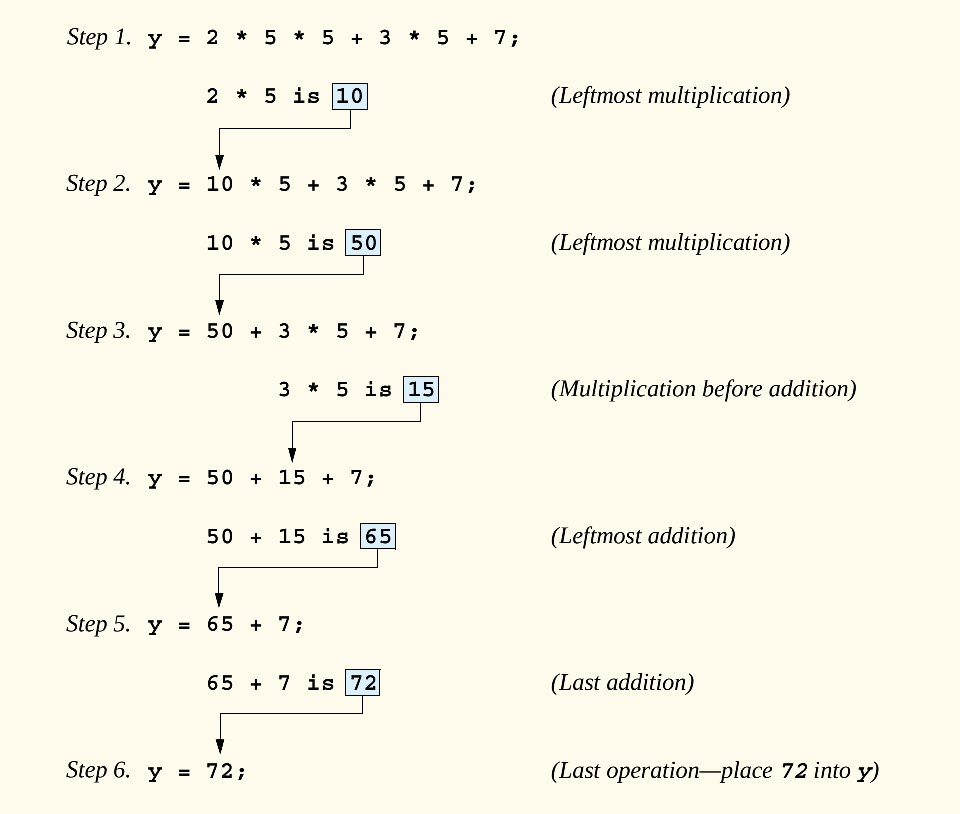

Step 1. y = 2 * 5 * 5 + 3 * 5 + 7;
2 * 5 is 10
(Leftmost multiplication)
Step 2. y = 10 * 5 + 3 * 5 + 7;
10 * 5 is 50
(Leftmost multiplication)
Step 3. y = 50 + 3 * 5 + 7;
3 * 5 is 15
(Multiplication before addition)
Step 4. y = 50 + 15 + 7;
50 + 15 is 65
(Leftmost addition)
Step 5. y = 65 + 7;
65 + 7 is 72
(Last addition)
Step 6. y = 72;
(Last operation—place 72 into y)

Fig. 3.17 Order in which a second-degree polynomial is evaluated.

3.6 Decision Making: Equality and Relational Operators

This section introduces C#'s **`if`** structure, which allows a program to make a decision based on the truth or falsity of some *condition*. If the condition is met (i.e., the condition is *true*), the statement in the body of the `if` structure executes. If the condition is not met (i.e., the condition is *false*), the body statement does not execute. Conditions in `if` structures can be formed by using the *equality operators* and *relational operators*, summarized in Fig. 3.18. The relational operators all have the same level of precedence and associate from left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right.

Common Programming Error 3.7

It is a syntax error if the operators `==`, `!=`, `>=` and `<=` contain spaces between their symbols (as in `= =`, `! =`, `> =`, `< =`).

Common Programming Error 3.8

Reversing the operators `!=`, `>=` and `<=` (as in `=!`, `=>` and `=<`) is a syntax error.

Standard algebraic equality operator or relational operator	C# equality or relational operator	Example of C# condition	Meaning of C# condition
Equality operators			
==	`==`	`x == y`	`x` is equal to `y`
≠	`!=`	`x != y`	`x` is not equal to `y`
Relational operators			
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`

Fig. 3.18 Equality and relational operators.

Common Programming Error 3.9

Confusing the equality operator `==` *with the assignment operator* `=` *is a logic error. The equality operator should be read "is equal to," and the assignment operator should be read "gets" or "gets the value of." Some people prefer to read the equality operator as "double equals" or "equals equals."*

The next example uses six `if` statements to compare two numbers input into a program by the user. If the condition in any of these `if` statements is true, the assignment statement associated with that `if` executes. The user inputs values that the program converts to integers and stores in variables `number1` and `number2`. The program compares the numbers and displays the results of the comparison in the command prompt. The program and sample outputs are shown in Fig. 3.19.

```
// Fig. 3.19: Comparison.cs
// Using if statements, relational operators and equality
// operators.

using System;

class Comparison
{
   static void Main( string[] args )
   {
      int number1,             // first number to compare
          number2;             // second number to compare

      // read in first number from user
      Console.Write( "Please enter first integer: " );
      number1 = Int32.Parse( Console.ReadLine() );

```

Fig. 3.19 Using equality and relational operators. (Part 1 of 2.)

```
      // read in second number from user
      Console.Write( "\nPlease enter second integer: " );
      number2 = Int32.Parse( Console.ReadLine() );

      if ( number1 == number2 )
         Console.WriteLine( number1 + " == " + number2 );

      if ( number1 != number2 )
         Console.WriteLine( number1 + " != " + number2 );

      if ( number1 < number2 )
         Console.WriteLine( number1 + " < " + number2 );

      if ( number1 > number2 )
         Console.WriteLine( number1 + " > " +  number2 );

      if ( number1 <= number2 )
         Console.WriteLine( number1 + " <= " + number2 );

      if ( number1 >= number2 )
         Console.WriteLine( number1 + " >= " + number2 );

   } // end method Main

} // end class Comparison
```

```
Please enter first integer: 2000

Please enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

```
Please enter first integer: 1000

Please enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Please enter first integer: 1000

Please enter second integer: 1000
1000 == 1000
1000 <= 1000
1000 >= 1000
```

Fig. 3.19 Using equality and relational operators. (Part 2 of 2.)

The definition of class `Comparison` begins on line 7, and the `Main` method begins on line 9. Lines 11–12 declare the variables used in method `Main`. Note that there are two variables of type `int`. Remember that variables of the same type may be declared in one declaration or in multiple declarations. Also recall that, if more than one variable is placed in one declaration (lines 11–12), those variables are separated by commas (`,`). The comment at the end of each line indicates the purpose of each variable in the program.

Line 16 reads in the first number from the user. Line 20 reads in the second number from the user. These values are stored in variables `number1` and `number2`, respectively. Recall that arithmetic operators cannot be used with strings. Relational and equality operators also cannot be used with strings. Therefore, the two input strings must be converted to integers.

Lines 16 and 20 both get an input, convert the input to type `int` and assign the values to the appropriate variable in one step. Notice that this step can be combined with the variable declaration and placed on one line with the statement

```
int number1 = Int32.Parse( Console.ReadLine() );
```

which declares the variable, reads a string from the user, converts the string to an integer and stores the integer in the variable.

The `if` structure in lines 22–23 compares the values of the variables `number1` and `number2` for equality. If the values are equal, the program outputs the value of `number1 + " == " + number2`. Notice that this expression uses the operator `+` to “add” (or combine) numbers and strings. C# has a version of the `+` operator used for ***`string`*** *concatenation*. Concatenation is the process that enables a `string` and a value of another data type (including another `string`) to be combined to form a new `string`.

If `number1` contains the value `1000` and `number2` contains the value `1000`, the expression evaluates as follows: C# determines that the operands of the `+` operator are of different types and that one of them is a `string`. Next, `number1` and `number2` are converted to a `string` and concatenated with `" == "`. At this point, the `string`, namely `"1000 == 1000"`, is sent to `Console.WriteLine` to be output. As the program proceeds through the `if` structures, more `string`s will be output by these `Console.WriteLine` statements. For example, given the value `1000` for `number1` and `number2`, the `if` conditions at lines 34 (`<=`) and 37 (`>=`) will also be true. Thus, the output displayed will be

```
1000 == 1000
1000 <= 1000
1000 >= 1000
```

The second of output window of Fig. 3.19 demonstrates this case.

Common Programming Error 3.10

Confusing the `+` operator used for string concatenation with the `+` operator used for addition can lead to strange results. For example, assuming integer variable `y` has the value `5`, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`. First the value of `y` (`5`) is concatenated with the string `"y + 2 = "`, then the value `2` is concatenated with the new, larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result.

Common Programming Error 3.11

Replacing operator `==` in the condition of an `if` structure, such as `if ( x == 1 )`, with operator `=`, as in `if ( x = 1 )`, is a logic error.

Notice the indentation in the **`if`** statements throughout the program. Such indentation enhances program readability.

Good Programming Practice 3.14

Indent the statement in the body of an **`if`** *structure to make the body of the structure stand out and to enhance program readability.*

Good Programming Practice 3.15

Place only one statement per line in a program. This enhances program readability.

Common Programming Error 3.12

Forgetting the left and right parentheses for the condition in an **`if`** *structure is a syntax error. The parentheses are required.*

There is no semicolon (**`;`**) at the end of the first line of each **`if`** structure. Such a semicolon would result in a logic error at execution time. For example,

```
if ( number1 == number2 );
   Console.WriteLine( number1 + " == " + number2 );
```

would actually be interpreted by C# as

```
if ( number1 == number2 )
   ;

Console.WriteLine( number1 + " == " + number2 );
```

where the semicolon on the line by itself—called the *empty statement*—is the statement to execute if the condition is true. When the empty statement executes, no task is performed. The program continues with the **`Console.WriteLine`** statement, which executes regardless of whether the condition is true or false.

Common Programming Error 3.13

Placing a semicolon immediately after the right parenthesis of the condition in an **`if`** *structure is normally a logic error. The semicolon causes the body of the* **`if`** *structure to be empty, so the* **`if`** *structure performs no action, regardless of whether its condition is true. Worse, the intended body statement of the* **`if`** *structure becomes a statement in sequence with the* **`if`** *structure and always executes.*

Notice the use of spacing in Fig. 3.19. Remember that the compiler normally ignores whitespace characters, such as tabs, newlines and spaces. Statements may be split over several lines and may be spaced according to the programmer's preferences without affecting the meaning of a program. It is incorrect to split identifiers and string literals. Ideally, statements should be kept small, but it is not always possible to do so.

Good Programming Practice 3.16

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines with one level of indentation.

The chart in Fig. 3.20 shows the precedence of the operators introduced in this chapter. The operators are displayed from top to bottom in decreasing order of precedence. Notice that all these operators, with the exception of the assignment operator `=`, associate from left to right. Addition is left associative, so an expression such as `x + y + z` is evaluated as if it were written `(x + y) + z`. The assignment operator `=` associates from right to left, so an expression such as `x = y = 0` is evaluated as if it were written `x = (y = 0)`. The latter expression, `x = (y = 0)`, first assigns the value `0` to variable `y` and then assigns the result of that assignment, `0`, to `x`.

Good Programming Practice 3.17

Refer to the operator-precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the expected order. If you are uncertain about the order of evaluation in a complex expression, use parentheses to force the order, as you would do in an algebraic expression. Remember that some operators, such as assignment (`=`), associate from right to left rather than from left to right.

In this chapter, we introduced important features of C#, including displaying data on the screen, inputting data from the keyboard, performing calculations and making decisions. The next chapter demonstrates many similar techniques, as we reintroduce C# Windows applications (applications that provide a graphical user interface). We also introduce *structured programming* and familiarize the reader further with indentation techniques. We study how to specify and vary the order in which statements execute—this order is called *flow of control*.

SUMMARY

- A console application is an application that, predominantly, displays text output in either a console window (or MS-DOS window). This is also called a command prompt.
- Programmers insert comments to document programs and improve program readability. Every program should begin with a comment describing the purpose of the program.
- A comment that begins with `//` is called a single-line comment, because the comment terminates at the end of the current line. A `//` comment can begin in the middle of a line and continue until that line's end. Multiple-line comments begin with the delimiter `/*` and end with delimiter `*/`. The compiler ignores all text between the delimiters of the comment.
- A namespace groups various C# features into related categories, providing programmers with the ability to locate them quickly.

Operators	**Associativity**	**Type**
`()`	left to right	parentheses
`*  /  %`	left to right	multiplicative
`+  -`	left to right	additive
`<  <=  >  >=`	left to right	relational
`==  !=`	left to right	equality
`=`	right to left	assignment

Fig. 3.20 Precedence and associativity of operators discussed in this chapter.

- The **`using`** directive declares the use of a namespace.
- Programmers use preexisting code to make programming easier and faster.
- Blank lines, space characters and tab characters are known as whitespace (space characters and tabs are known, specifically, as whitespace characters). Such characters are ignored by the compiler and used to improve program readability.
- Classes consist of pieces (called methods) that perform tasks and return information (or simply control) when they complete their tasks. The programmer can program each piece that is needed to form a C# program.
- Classes defined by the programmer are known as programmer-defined or user-defined classes.
- The **`class`** keyword introduces a class definition and is followed immediately by the class name.
- Keywords are reserved for use by C# and are always spelled with lowercase letters.
- By convention, all class names in C# begin with an uppercase letter and have an uppercase letter for the beginning of every word in the class name.
- The name of a class is called an identifier. An identifier is a series of characters, consisting of letters, digits, underscores (_) and "at" symbols (**`@`**), that does not begin with a digit and does not contain any spaces.
- C# is case sensitive—uppercase and lowercase letters are different, thus **`a1`** and **`A1`** are distinct identifiers.
- A left brace (**`{`**) begins the body of every class or method definition. A corresponding right brace (**`}`**) must end each class or method definition. If braces do not occur in matching pairs, the compiler indicates an error.
- Set a convention for the indent size you prefer and apply that convention uniformly.
- C# applications begin executing at **`Main`**, which is known as the entry point of the program.
- C# class definitions normally contain one or more methods. C# applications contain one or more classes. For a C# application, one of the classes in the application must contain method **`Main`**.
- Methods can perform tasks and return information when these tasks complete. Information also can be passed to a method. This information may be necessary for the method to complete its task and is called an argument.
- A string sometimes is called a character string, a message or a string literal.
- Whitespace characters in strings are not ignored by the compiler.
- Every statement must end with a semicolon (the statement terminator). Omitting the semicolon at the end of a statement is a syntax error.
- A syntax error occurs when the compiler cannot recognize a statement. The compiler normally issues an error message to help the programmer locate and fix the incorrect statement. Syntax errors are violations of the language's rules.
- When the compiler reports a syntax error, the error might not be on the line indicated by the error message. First, check the line where the error was reported. If that line does not contain syntax errors, check the preceding several lines in the program.
- Unlike **`WriteLine`**, method **`Write`** does not position the output cursor at the beginning of the next line in the console window after displaying its argument.
- A single statement can display multiple lines by using newline characters.
- C# has a version of the **`+`** operator for string concatenation that enables a string and a value of another data type (including another string) to be concatenated—the result of this operation is a new (and normally longer) string.

- The backslash (\) is called an escape character. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence.
- String contents always must be delimited with double quotes.
- Class `MessageBox` allows you to display a dialog containing information.
- Class `MessageBox` is defined in namespace `System.Windows.Forms`.
- The predefined namespaces in C# contain classes that are collectively referred to as the .NET Framework Class Library.
- GUI components facilitate data entry by the user and the formatting or presenting of data outputs to the user.
- Method `MessageBox.Show` is a special method of class `MessageBox`, called a static method. Such methods are always called with their class name followed by a dot operator (`.`) and the method name.
- Depending on the type of application we create, classes may be compiled into files with a `.exe` (executable) extension, a `.dll` (or dynamic link library) extension or one of several other extensions. This file is called an assembly, which is the packaging unit for code in C#.
- We need to add a reference to an assembly if we wish to use its classes. References to assemblies can be created easily in Visual Studio .NET by selecting the **Add Reference…** option from the **Project** menu and finding the necessary `.dll`.
- The `System.Windows.Forms` namespace contains many classes that help C# programmers define graphical user interfaces (GUIs) for their applications.
- A message dialog by default includes an **OK** button that allows the user to dismiss the dialog.
- A variable is a location in memory where a value can be stored for use by a program.
- All variables must be declared with a name and a data type before they can be used in a program.
- A variable name can be any valid identifier.
- Declarations end with a semicolon (`;`) and can be split over several lines, with each variable in the declaration separated by a comma.
- Several variables of the same type may be declared in either one declaration or separate declarations.
- The keywords `int`, `double` and `char` are primitive types.
- Primitive type names are keywords.
- A prompt is a message that directs the user to take a specific action.
- The `=` operator is called a binary operator, because it has two operands. A statement containing an `=` operation is called an assignment statement, because it assigns a value to a variable. The expression to the right side of the assignment operator `=` is always evaluated before the assignment occurs.
- Method `Int32.Parse` (a static method of class `Int32`) converts its `string` argument to an integer.
- Sometimes, when displaying strings C# encounters a format. A format specifies a placeholder for a value that will be inserted in a string.
- Variable names actually correspond to locations in the computer's memory. Every variable has a name, a type, a size and a value.
- Whenever a value is placed in a memory location, this value replaces the previous value in that location. The previous value is destroyed (lost).
- When a value is read from a memory location, the process is nondestructive.

- Integer division yields an integer quotient. Note that any fractional part in integer division is simply discarded (i.e., truncated)—no rounding occurs.
- The modulus operator (`%`) yields the remainder after integer division.
- Arithmetic expressions must be written in straight-line form to facilitate entering programs into the computer.
- Parentheses are used in C# expressions in the same manner as in algebraic expressions.
- C# applies the operators in arithmetic expressions in a precise sequence determined by the rules of operator precedence.
- As in algebra, it is acceptable to place unnecessary (redundant) parentheses in an expression to make the expression clearer.
- The `if` structure allows a program to make a decision based on the truth or falsity of some condition. If the condition is met (i.e., the condition is true), the statement in the body of the `if` structure executes. If the condition is not met (i.e., the condition is false), the body statement does not execute.
- Conditions in `if` structures can be formed by using equality operators and relational operators.
- A string containing no characters is known as an empty string.
- Every variable declared in a method must be initialized (given a value) before it can be used in an expression, or a syntax error will occur.
- A semicolon by itself (not preceded by an actual statement) is known as an empty statement. When an empty statement executes, no task is performed.

TERMINOLOGY

`!=` is-not-equal-to operator
`"` double quotation
`%` modulus operator
`*/` end a multiline comment
`/*` start a multiline comment
`//` single-line comment
`;` statement terminator
`<` less-than operator
`<=` less-than-or-equal-to operator
`=` assignment operator
`==` is-equal-to operator
`>` is-greater-than operator
`>=` greater-than-or-equal-to operator
`\\` escape sequence
`\n` escape sequence
`\r` escape sequence
`\t` escape sequence
`_` underscore
`{` left brace
`}` right brace
`,` comma
Add Reference dialog
algebraic notation
application
argument
arithmetic calculation
arithmetic operators
assembly
assignment statement
associativity of operators
asterisk (`*`) indicating multiplication
average
backslash (`\`)
binary operator
blank line
body of a class definition
body of a method definition
built-in data type
button
C# compiler
carriage return
case sensitive
`char` variable
character set
character string
class
class definition
`class` keyword
class name
comma (`,`)
command prompt
console window

comma-separated list of variable names
comment
compiler
compile-time error
concatenation of **`string`**s
condition
console application
`Console` class
`Console.ReadLine` method
console window
`Console.Write` method
`Console.WriteLine` method
data type
decision
declaration
dialog
display output
documentation
dot (**`.`**) operator
`double`
embedded parentheses
empty statement (**`;`**)
Enter (or *Return*) key
entry point of a program
error handling
escape sequence
exponentiation
`float`
flow of control
format
formatting strings
identifier
`if` structure
indentation in **`if`** statements
indentation techniques
innermost pair of parentheses
inputting data from the keyboard
`Int32.Parse` method
integer division
integer quotient
keyboard
keyword
left-to-right evaluation
location in the computer's memory
logic error
`Main` method
making decisions
matching left and right braces
`MessageBox` class
method
method definition
MS-DOS prompt
MSIL (Microsoft Intermediate Language)
multiple-line comment (**`/*`**… **`*/`**)
name of a variable
namespace
nested parentheses
.NET Framework Class Library
nondestructive
object
OK button on a dialog
operand
operator precedence
output
parentheses **`()`**
parentheses "on the same level"
`Parse` method
performing a calculation
polynomial
precedence
primitive data type
programmer-defined class
prompt
`ReadLine` method
real number
redundant parentheses
"reinventing the wheel"
reserved word
reuse
robust
rounding
run-time logic error
self-documenting code
single-line comment
size of a variable
space character
spacing convention
special character
standard output
statement
`static` method
straight-line form
string
`string` concatenation
string formatting
string literal
string of characters
`string` type
structured programming
syntax error

`System` namespace
`System.Windows.Forms` namespace
tab character
text editor
truncate
type of a variable
Unicode
unnecessary parentheses
user-defined class
`using` directive
value of a variable
variable
Visual Studio .NET-generated console application
`void` keyword
whitespace character
Windows 95/98
Windows application
Windows NT/2000
`Write` method of class `Console`
`WriteLine` method of class `Console`

SELF-REVIEW EXERCISES

3.1 Fill in the blanks in each of the following statements:

a) The __________ and __________ begin and end every method body.
b) Every statement must end with a __________ statement terminator.
c) The __________ structure makes decisions.
d) __________ begins a single-line comment.
e) __________, __________, __________ and __________ are known as whitespace.
f) Class __________ displays message dialogs.
g) __________ are reserved for use by C#.
h) C# applications begin execution at method __________.
i) Methods __________ and __________ display information in the console window.
j) A __________ method is invoked by preceding its name with its class name and a dot (`.`).
k) A __________ begins the body of a method definition.
l) A C# program includes __________ directives to indicate that we are incorporating classes from certain namespaces.
m) When a value is placed in a memory location, this value __________ the previous value in that location.
n) Saying that operators are applied from left to right refers to the __________ of the operators.
o) C#'s `if` structure allows a program to make a decision based on the __________ or __________ of a condition.
p) Types such as `int`, `float`, `double` and `char` are often called __________ data types.
q) A variable is a location in the computer's __________ where a value can be stored .
r) Data types __________ and __________ contain decimal points for storing numbers such as 3.44 or 1.20846.
s) The expression to the __________ of the assignment operator (`=`) is always evaluated first.
t) Arithmetic expressions in C# must be written in __________ form to facilitate entering programs into the computer.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Comments cause the computer to print the text after the `//` on the screen when the program is executed.
b) All variables must be given a type when they are declared.
c) C# considers the variables `number` and `NuMbEr` to be identical.
d) The arithmetic operators `*`, `/`, `%`, `+` and `-` all have the same level of precedence.
e) Method `Int32.Parse` converts an integer to a `string`.
f) A comment that begins with `//` is called a single-line comment.
g) A string of characters contained between double quotation marks is called a phrase or phrase literal.
h) Blank lines, space characters, newline characters and tab characters are ignored by the compiler when placed outside strings.

- i) Every C# application must contain one `Main` method.
- j) Curly braces that define bodies of classes and methods need not occur in matching pairs.
- k) C# applications begin executing at `Main`.
- l) The compiler uses `class` statements to identify namespaces referenced in a C# program.
- m) Integer division yields an integer quotient.
- n) Parentheses cannot be used in an arithmetic expression to force the order of evaluation of operators to occur in a sequence determined by the programmer.

ANSWERS TO SELF-REVIEW EXERCISES

3.1 a) Left brace (`{`), right brace (`}`). b) Semicolon (`;`). c) `if`. d) `//`. e) Blank lines, space characters, newline characters and tab characters. f) `MessageBox`. g) Keywords. h) `Main`. i) `Console.Write` and `Console.WriteLine`. j) static. k) left brace. l) `using`. m) replaces. n) associativity. o) truth, falsity. p) primitive (or built-in). q) memory. r) `float`, `double`. s) right. t) straight-line.

3.2 a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs and improve their readability. b) True. c) False. C# is case sensitive, so these variables are distinct. d) False. The operators `*`, `/` and`%` are on the same level of precedence, and the operators `+` and `-` are on a lower level of precedence. e) False. Method `Integer.Parse` converts a `string` to an integer (`int`) value. f) True. g) False. A string of characters is called a string or string literal. h) True. i) True. j) False. Curly braces that do not match cause syntax errors. k) True. l) False. The compiler uses `using` directives to identify and load namespaces. m) True. n) False. Parentheses can be used to force the order of evaluation.

EXERCISES

3.3 Write C# statements that accomplish each of the following tasks:

- a) Display the message `"Enter two numbers"`, using class `MessageBox`.
- b) Assign the product of variables `b` and `c` to variable `a`.
- c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).

3.4 What displays in the message dialog when each of the following C# statements is performed? Assume the value of `x` is 2 and the value of `y` is 3.

- a) `MessageBox.Show( "x = " + x );`
- b) `MessageBox.Show( "The value of x + x is " + ( x + x ) );`
- c) `MessageBox.Show( "x =" );`
- d) `MessageBox.Show( ( x + y ) + " = " + ( y + x ) );`

3.5 Given $y = ax^3 + 7$, which of the following are correct statements for this equation?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * ( x + 7 );`
- c) `y = ( a * x ) * x * ( x + 7 );`
- d) `y = ( a * x ) * x * x + 7;`
- e) `y = a * ( x * x * x ) + 7;`
- f) `y = a * x * ( x * x + 7 );`

3.6 Indicate the order of evaluation of the operators in each of the following C# statements, and show the value of `x` after each statement is performed.

- a) `x = 7 + 3 * 6 / 2 - 1;`
- b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
- c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

3.7 Write an application that displays the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Write the program using the following methods:

a) Use one `Console.Write` statement.
b) Use four `Console.Write` statements.

3.8 Write an application that asks the user to enter two numbers, obtains the two numbers from the user and prints the sum, product, difference and quotient of the two numbers.

3.9 Write an application that inputs from the user the radius of a circle and prints the circle's diameter, circumference and area. Use the following formulas (r is the radius): $diameter = 2r$, $circumference = 2\pi r$, $area = \pi r^2$.

3.10 Write an application that displays in the console window a box, an oval, an arrow and a diamond, using asterisks (`*`) as follows:

```
*********          ***              *              *
*       *        *     *           ***            * *
*       *       *       *         *****          *   *
*       *       *       *           *           *     *
*       *       *       *           *          *       *
*       *       *       *           *           *     *
*       *       *       *           *            *   *
*       *        *     *            *             * *
*********          ***              *              *
```

3.11 Modify the program you created in Exercise 3.12 to display the shapes in a `MessageBox` dialog. Does the program display the shapes exactly as in Exercise 3.12?

3.12 What does the following code print?

```
Console.WriteLine( "*\n**\n***\n****\n*****" );
```

3.13 What does the following code print?

```
Console.Write( "*" );
Console.Write( "***" );
Console.WriteLine( "*****" );
Console.Write( "****" );
Console.WriteLine( "**" );
```

3.14 Write an application that reads in two integers and determines and prints whether the first is a multiple of the second. For example, if the user inputs 15 and 3, the first number is a multiple of the second. If the user inputs 2 and 4, the first number is not a multiple of the second. [*Hint*: Use the modulus operator.]

3.15 Here is a peek ahead. In this chapter, you learned about integers and the data type `int`. C# can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. Every character has a corresponding integer representation. The set of characters a computer uses and the corresponding integer representations for those characters is called that computer's character set. You can indicate a character value in a program simply by enclosing that character in single quotes, as with `'A'`.

You can determine the integer equivalent of a character by preceding that character with `(int)`—this is called a cast. (We will say more about casts in Chapter 4.)

```
( int ) 'A'
```

The following statement would output a character and its integer equivalent:

```
Console.WriteLine(
   "The character " + 'A' + " has the value " + (int) 'A' );
```

When the preceding statement executes, it displays the character **`A`** and the value **`65`** (from the Unicode character set) as part of the string.

Write an application that displays the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. As a minimum, display the integer equivalents of the following: **`A B C a b c 0 1 2 $ * + /`** and the blank character.

3.16 Write an application that inputs one number consisting of five digits from the user, separates the number into its individual digits and prints the digits separated from one another by three spaces each. For example, if the user types in the number **`42339`**, the program should print

```
4   2   3   3   9
```

[*Hint*: This exercise is possible with the techniques you learned in this chapter. You will need to use both division and modulus operations to “pick off” each digit.]

For the purpose of this exercise, assume that the user enters the correct number of digits. What happens when you execute the program and type a number with more than five digits? What happens when you execute the program and type a number with fewer than five digits?

3.17 Using only the programming techniques you learned in this chapter, write an application that calculates the squares and cubes of the numbers from 0 to 10 and prints the resulting values in table format, as follows:

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

[*Note*: This program does not require any input from the user.]

3.18 Write a program that reads a first name and a last name from the user as two separate inputs and concatenates the first name and last name, but separated by a space. Display the concatenated name at the command prompt.

3.19 (*“The Twelve Days of Christmas” Song*) Write an application that uses repetition and **`if`** structures to print the song “The Twelve Days of Christmas.” Visit the Web site **`www.12days.com/library/carols/12daysofxmas.htm`** for the complete lyrics to the song.

4

Control Structures: Part 1

Objectives

- To understand basic problem-solving techniques of programming.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the **`if`** and **`if`**/**`else`** selection structures to choose among alternative actions.
- To use the **`while`** repetition structure to execute statements in a program repeatedly.
- To understand counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and assignment operators.

Let's all move one place on.
Lewis Carroll

The wheel is come full circle.
William Shakespeare, *King Lear*

How many apples fell on Newton's head before he took the hint?
Robert Frost, Comment

Outline

4.1 Introduction
4.2 Algorithms
4.3 Pseudocode
4.4 Control Structures
4.5 `if` Selection Structure
4.6 `if/else` Selection Structure
4.7 `while` Repetition Structure
4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)
4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)
4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)
4.11 Assignment Operators
4.12 Increment and Decrement Operators
4.13 Introduction to Windows Application Programming

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

4.1 Introduction

Before writing a program to solve a problem, it is essential to have a thorough understanding of the problem and a carefully planned approach. When writing a program, it is equally essential to understand the types of building blocks that are available and to employ proven program construction principles. In this chapter and the next, we present the theory and principles of structured programming. The techniques you will learn are applicable to most high-level languages, including C#. When we study object-based programming in more depth in Chapter 8, we will see that control structures are helpful in building and manipulating objects. The control structures discussed in this chapter will enable you to build these objects in a quick and easy manner.

4.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A *procedure* for solving a problem in terms of

1. the *actions* to be executed and
2. the *order* in which these actions are to be executed

is called an *algorithm*. The example that follows demonstrates the importance of correctly specifying the order in which the actions are to be executed.

Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work well-prepared to make critical decisions.

Suppose that the same steps are performed in a slightly different order: (1) get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our executive shows up for work soaking wet.

The importance of correctly specifying the order in which actions appear applies to computer programs, as well. *Program control* refers to the task of ordering a program's statements correctly. In this chapter, we begin to investigate the program control capabilities of C#.

4.3 Pseudocode

Pseudocode is an artificial and informal language that helps programmers develop algorithms. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of C# programs. Pseudocode is similar to everyday English; it is convenient and user-friendly, and it is not an actual computer programming language.

Pseudocode is not executed on computers. Rather, pseudocode helps the programmer "think out" a program before attempting to write it in a programming language, such as C#. In this chapter, we provide several examples of pseudocode algorithms.

Software Engineering Observation 4.1

Pseudocode helps the programmer conceptualize a program during the program design process. The pseudocode may then be converted to C#.

The style of pseudocode that we present consists solely of characters, thus programmers may type pseudocode conveniently using an editor program. Programmers can convert carefully prepared pseudocode programs to corresponding C# programs easily. In many cases, this conversion takes place simply by replacing pseudocode statements with their C# equivalents.

Pseudocode normally describes only executable statements—the actions that are performed when the pseudocode is converted to C# and executed. Declarations are not executable statements. For example, the declaration

```
int i;
```

informs the compiler of the type of variable `i` and instructs the compiler to reserve space in memory for this variable. This declaration does not cause any action, such as input, output or a calculation, to occur when the program executes. Some programmers choose to list variables and their purposes at the beginning of a pseudocode program.

4.4 Control Structures

Normally, statements in a program execute one after the other in the order in which they appear in the program. This is called *sequential execution*. Various C# statements enable the programmer to specify that the next statement to execute may not be the next one in sequence. A *transfer of control* occurs when a statement other than the next one in the program executes.

During the 1960s, it became clear that the indiscriminate use of transfers of control was causing difficulty for software development groups. The problem was the `goto` *statement*, which, in some programming languages, allows the programmer to specify a transfer of control to one of a wide range of possible destinations in a program. This caused programs to become quite unstructured and hard to follow. The notion of *structured programming* became almost synonymous with "`goto` elimination."

The research of Bohm and Jacopini[1] demonstrated that all programs with `goto` statements could be written without them. The challenge of the era for programmers was to shift their styles to "`goto`-less programming." It was not until the 1970s that programmers started taking structured programming seriously. The results were impressive, as software development groups reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify and more likely to be bug-free in the first place.

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three *control structures*, namely, the *sequence structure*, the *selection structure* and the *repetition structure*. The sequence structure is built into C#. Unless directed otherwise, the computer executes C# statements one after the other in the order in which they appear in a program. The *flowchart* segment of Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order.

A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts contain certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles. These symbols are connected by arrows called *flowlines*, which indicate the order in which the actions of the algorithm execute. This order is known as the flow of control.

Like pseudocode, flowcharts often are useful for developing and representing algorithms, although pseudocode is preferred by many programmers. Flowcharts show clearly how control structures operate; that is all we use them for in this text. The reader should compare carefully the pseudocode and flowchart representations of each control structure.

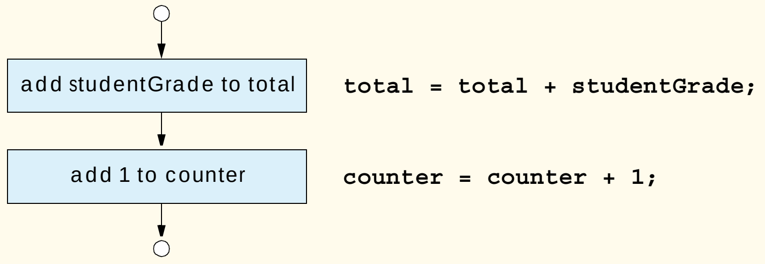


Fig. 4.1 Flowcharting C#'s sequence structure.

---

[1] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

Consider the flowchart segment for the sequence structure in Fig. 4.1. We use the *rectangle symbol*, also called the *action symbol*, to indicate any type of action, including a calculation or an input/output operation. The flowlines in the figure indicate the order in which the actions are to be performed—first, `studentGrade` is to be added to `total`, then `1` is to be added to `counter`. We can have as many actions as we want in a sequence structure. Anywhere in a sequence that a single action may be placed, several actions may also be placed.

When drawing a flowchart that represents a complete algorithm, an *oval symbol* containing the word “Begin” is the first symbol used; an oval symbol containing the word “End” indicates where the algorithm ends. When drawing only a portion of an algorithm, as in Fig. 4.1, the oval symbols are omitted in favor of using *small circle symbols*, also called *connector symbols*.

Perhaps the most important flowcharting symbol is the *diamond symbol*, also called the *decision symbol*, which indicates that a decision is to be made. We discuss the diamond symbol in Section 4.5.

C# provides three types of selection structures, which we discuss in this chapter and the next. The `if` selection structure performs (selects) an action if a condition is true or skips the action if the condition is false. The `if`/`else` selection structure performs an action if a condition is true and performs a different action if the condition is false. The `switch` selection structure, discussed in Chapter 5, Control Structures: Part 2, performs one of many actions, depending on the value of an expression.

The `if` structure is called a *single-selection structure* because it selects or ignores a single action (or a single group of actions). The `if`/`else` structure is called a *double-selection structure* because it selects between two different actions (or groups of actions). The `switch` structure is called a *multiple-selection structure* because it selects among many different actions (or groups of actions).

C# provides four repetition structures—`while`, `do`/`while`, `for` and `foreach` (`while` is covered in this chapter, `do`/`while` and `for` are covered in Chapter 5, Control Structures: Part 2, and `foreach` is covered in Chapter 8, Object-Based Programming). Each of the words `if`, `else`, `switch`, `while`, `do`, `for` and `foreach` are C# keywords. Figure 4.2 lists the complete set of C# keywords. We discuss the vast majority of C#'s keywords throughout this book.

C# Keywords				
`abstract`	`as`	`base`	`bool`	`break`
`byte`	`case`	`catch`	`char`	`checked`
`class`	`const`	`continue`	`decimal`	`default`
`delegate`	`do`	`double`	`else`	`enum`
`event`	`explicit`	`extern`	`false`	`finally`
`fixed`	`float`	`for`	`foreach`	`goto`
`if`	`implicit`	`in`	`int`	`interface`

Fig. 4.2 C# keywords. (Part 1 of 2.)

C# Keywords				
`internal`	`is`	`lock`	`long`	`namespace`
`new`	`null`	`object`	`operator`	`out`
`override`	`params`	`private`	`protected`	`public`
`readonly`	`ref`	`return`	`sbyte`	`sealed`
`short`	`sizeof`	`stackalloc`	`static`	`string`
`struct`	`switch`	`this`	`throw`	`true`
`try`	`typeof`	`uint`	`ulong`	`unchecked`
`unsafe`	`ushort`	`using`	`virtual`	`void`
`volatile`	`while`			

Fig. 4.2 C# keywords. (Part 2 of 2.)

C# has only eight control structures—sequence, three types of selection and four types of repetition. Each program is formed by combining as many of each type of control structure as is necessary. As with the sequence structure in Fig. 4.1, each control structure is flowcharted with two small circle symbols, one at the entry point to the control structure and one at the exit point.

Single-entry/single-exit control structures make it easy to build programs—the control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This is similar to the stacking of building blocks; thus, we call it *control-structure stacking*. There is only one other way control structures may be connected, and that is through *control-structure nesting*, where one control structure can be placed inside another. Thus, algorithms in C# programs are constructed from only eight different types of control structures combined in only two ways.

4.5 `if` Selection Structure

In a program, a selection structure chooses among alternative courses of action. For example, suppose that the passing grade on an examination is 60 (out of 100). Then the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

determines if the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, then *Passed* is printed, and the next pseudocode statement in order is "performed." (Remember that pseudocode is not a real programming language.) If the condition is false, the print statement is ignored, and the next pseudocode statement in order is performed. Note that the second line of this selection structure is indented. Such indentation is optional, but it is highly recommended because it emphasizes the inherent structure of structured programs. The preceding pseudocode *If* statement may be written in C# as

```
if ( studentGrade >= 60 )
   Console.WriteLine( "Passed" );
```

Notice that the C# code corresponds closely to the pseudocode, demonstrating how pseudocode can be useful as a program development tool. The statement in the body of the `if` structure outputs the character string `"Passed"` in the console window.

The flowchart in Fig. 4.3 illustrates the single-selection `if` structure. This flowchart contains the most important flowcharting symbol—the decision (or diamond) symbol, which indicates that a decision is to be made. The decision symbol contains a condition, that can be either `true` or `false`. The decision symbol has two flowlines emerging from it. One indicates the direction to be taken when the condition in the symbol is true; the other indicates the direction to be taken when the condition is false. A decision can be made on any expression that evaluates to a value of C#'s ***`bool`*** type (i.e., any expression that evaluates to `true` or `false`).

Note that the `if` structure, too, is a single-entry/single-exit structure. The flowcharts for the remaining control structures also contain (aside from small circle symbols and flowlines) only rectangle symbols, to indicate the actions to be performed, and diamond symbols, to indicate decisions to be made. This is the *action/decision model of programming* we have been emphasizing.

We can envision eight bins, each containing control structures for only one of the eight types. The control structures in each bin are empty; nothing is written in the rectangles or diamonds. The programmer's task is to assemble a program using as many control structures as the algorithm demands, combining those control structures in only two possible ways (stacking or nesting), then filling in the actions and decisions in a manner appropriate for the algorithm. We will discuss the variety of ways in which actions and decisions may be written.

4.6 `if`/`else` Selection Structure

The `if` selection structure performs an indicated action only when the condition evaluates to true; otherwise, the action is skipped. The `if`/`else` selection structure allows the programmer to specify different actions to perform when the condition is true and when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

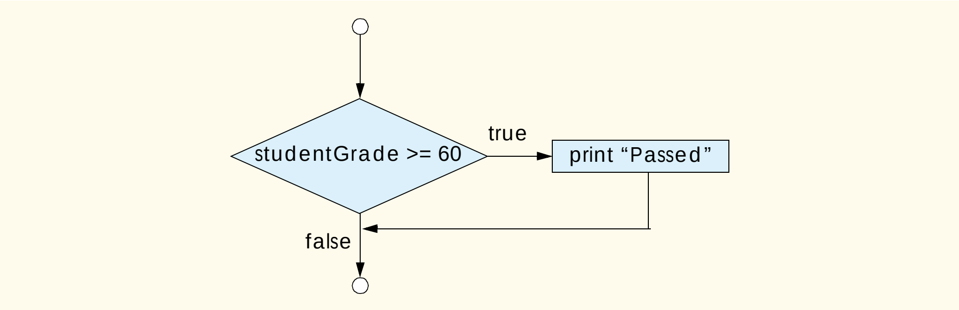


Fig. 4.3 Flowcharting a single-selection `if` structure.

prints *Passed* if the student's grade is greater than or equal to 60, and prints *Failed* if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding pseudocode *If/Else* structure may be written in C# as

```
if ( studentGrade >= 60 )
   Console.WriteLine( "Passed" );
else
   Console.WriteLine( "Failed" );
```

Good Programming Practice 4.1

Indent both body statements of an **`if`**/**`else`** *structure.*

Note that the body of the **`else`** statement also is indented. The indentation convention you choose should be applied carefully throughout your programs. It is difficult to read programs that do not use uniform spacing conventions.

The flowchart in Fig. 4.4 illustrates the flow of control in the **`if`**/**`else`** structure. Note that (besides small circles and arrows) the only symbols in the flowchart are rectangles (for actions) and a diamond (for a decision). We continue to emphasize this action/decision model of computing.

The *conditional operator* (**`?:`**) is related closely to the **`if`**/**`else`** structure. The **`?:`** is C#'s only *ternary operator*—it takes three operands. The operands and the **`?:`** form a *conditional expression*. The first operand is a *condition* (i.e., an expression that evaluates to a **`bool`** value), the second is the value for the conditional expression if the condition evaluates to **`true`** and the third is the value for the conditional expression if the condition evaluates to **`false`**. For example, the output statement

```
Console.WriteLine( studentGrade >= 60 ? "Passed" : "Failed" );
```

contains a conditional expression that evaluates to the string **`"Passed"`** if the condition **`studentGrade >= 60`** is true and evaluates to the string **`"Failed"`** if the condition is false.

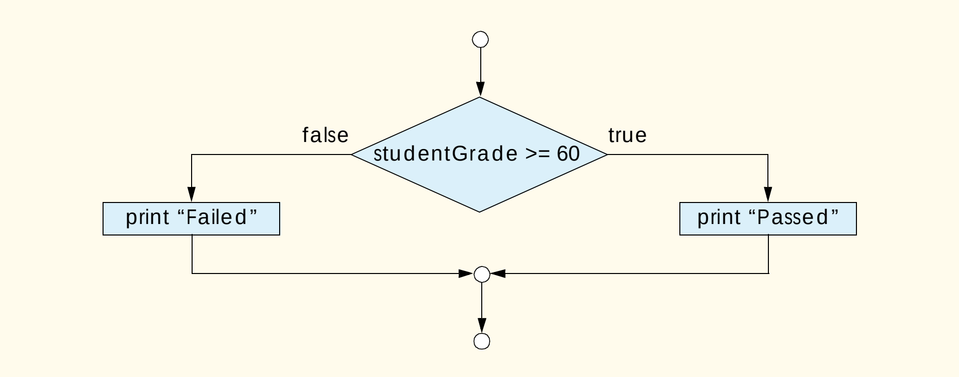


Fig. 4.4 Flowcharting a double-selection **`if`**/**`else`** structure.

The statement with the conditional operator performs in the same manner as the preceding `if`/`else` statement. The precedence of the conditional operator is low, so the entire conditional expression normally is placed in parentheses. Conditional operators can be used in some situations where `if`/`else` statements cannot, such as the argument to the `WriteLine` method shown earlier.

Nested ***`if`/`else`*** *structures* can test for multiple cases by placing `if`/`else` structures inside other `if`/`else` structures. For example, the following pseudocode statement will print `A` for exam grades greater than or equal to 90, `B` for grades in the range 80–89, `C` for grades in the range 70–79, `D` for grades in the range 60–69 and `F` for all other grades:

```
If student’s grade is greater than or equal to 90
    Print “A”
Else
    If student’s grade is greater than or equal to 80
        Print “B”
    Else
        If student’s grade is greater than or equal to 70
            Print “C”
        Else
            If student’s grade is greater than or equal to 60
                Print “D”
            Else
                Print “F”
```

This pseudocode may be written in C# as

```
if ( studentGrade >= 90 )
   Console.WriteLine( "A" );
else
   if ( studentGrade >= 80 )
      Console.WriteLine( "B" );
   else
      if ( studentGrade >= 70 )
         Console.WriteLine( "C" );
      else
         if ( studentGrade >= 60 )
            Console.WriteLine( "D" );
         else
            Console.WriteLine( "F" );
```

If `studentGrade` is greater than or equal to 90, the first four conditions are true, but only the `Console.WriteLine` statement after the first test executes. After that particular `Console.WriteLine` executes, the program skips the `else` part of the “outer” `if`/`else` structure.

Good Programming Practice 4.2

If there are several levels of indentation, each level should be indented the same additional amount of space.

Most C# programmers prefer to write the preceding `if` structure as

```
if ( studentGrade >= 90 )
   Console.WriteLine( "A" );
else if ( studentGrade >= 80 )
   Console.WriteLine( "B" );
else if ( studentGrade >= 70 )
   Console.WriteLine( "C" );
else if ( studentGrade >= 60 )
   Console.WriteLine( "D" );
else
   Console.WriteLine( "F" );
```

Both forms are equivalent. The latter form is popular because it avoids the deep indentation of the code. Such indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

The C# compiler always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`). This is referred to as the *dangling-else problem*. For example,

```
if ( x > 5 )
   if ( y > 5 )
      Console.WriteLine( "x and y are > 5" );
else
   Console.WriteLine( "x is <= 5" );
```

appears to indicate that if `x` is greater than `5`, the `if` structure in its body determines if `y` is also greater than `5`. If so, the string `"x and y are > 5"` is output. Otherwise, it *appears* that if `x` is not greater than `5`, the `else` part of the `if`/`else` structure outputs the string `"x is <= 5"`.

Testing and Debugging Tip 4.1

The reader can use Visual Studio to indent code properly. In order to check indentation, the reader should highlight the relevant code and press Ctrl-K *followed immediately by* Ctrl-F.

However, the preceding nested `if` structure does not execute as its indentation implies. The compiler actually interprets the structure as

```
if ( x > 5 )
   if ( y > 5 )
      Console.WriteLine( "x and y are > 5" );
   else
      Console.WriteLine( "x is <= 5" );
```

in which the body of the first `if` structure is an `if`/`else` structure. This structure tests if `x` is greater than `5`. If so, execution continues by testing if `y` is also greater than `5`. If the second condition is true, the proper string—`"x and y are > 5"`—is displayed. However, if the second condition is false, the string `"x is <= 5"` is displayed, even though we know `x` is greater than `5`.

To force the preceding nested `if` structure to execute as it was originally intended, the structure must be written as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      Console.WriteLine( "x and y are > 5" );
}
else
   Console.WriteLine( "x is <= 5" );
```

The braces (`{}`) indicate to the compiler that the second `if` structure is in the body of the first `if` structure and that the `else` is matched with the first `if` structure.

The `if` selection structure normally expects only one statement in its body. To include several statements in the body of an `if`, enclose these statements in braces (`{` and `}`). A set of statements contained in a pair of braces is called a *block*.

Software Engineering Observation 4.2

A block can be placed anywhere in a program at which a single statement can be placed.

The following example includes a block in the `else` part of an `if`/`else` structure:

```
if ( studentGrade >= 60 )
   Console.WriteLine( "Passed" );
else
{
   Console.WriteLine( "Failed" );
   Console.WriteLine( "You must take this course again." );
}
```

In this case, if `studentGrade` is less than 60, the program executes both statements in the body of the `else` and prints

```
Failed
You must take this course again.
```

Notice the braces surrounding the two statements in the `else` clause. These braces are important. Without the braces, the statement

```
Console.WriteLine( "You must take this course again." );
```

would be outside the body of the `else` and would execute regardless of whether the grade is less than 60.

Common Programming Error 4.1

Forgetting one of the braces that delimit a block can lead to syntax errors. Forgetting both of the braces that delimit a block can lead to syntax and/or logic errors.

Syntax errors, such as when one brace in a block is left out of the program, are caught by the compiler. A *logic error*, such as the error caused when both braces in a block are left out of the program, has its effect at execution time. A *fatal logic error* causes a program to fail and terminate prematurely. A *nonfatal logic error* allows a program to continue executing, but the program produces incorrect results.

Software Engineering Observation 4.3

Just as a block can be placed anywhere a single statement can be placed, it is also possible to have an empty statement, which is represented by placing a semicolon (`;`) where a statement normally would be.

Common Programming Error 4.2

Placing a semicolon after the condition in an **`if`** *structure leads to a logic error in single-selection* **`if`** *structures and a syntax error in double-selection* **`if`** *structures (if the* **`if`** *clause contains a nonempty body statement).*

Good Programming Practice 4.3

Some programmers prefer to type the beginning and ending braces of blocks before typing the individual statements within the braces. This practice helps avoid omitting one or both of the braces.

In this section, we introduced the notion of a block. A block may contain declarations. The declarations in a block commonly are placed first in the block before any action statements, but declarations may be intermixed with action statements.

4.7 `while` Repetition Structure

A *repetition structure* allows the programmer to specify that an action is to be repeated while a condition remains true. The pseudocode statement

```
While there are more items on my shopping list
    Purchase next item and cross it off my list
```

describes the repetition that occurs during a shopping trip. The condition, “there are more items on my shopping list” may be true or false. If it is true, then the action, “Purchase next item and cross it off my list” is performed. This action executes repeatedly while the condition remains true. The statement(s) contained in the *while* repetition structure constitute the body of the *while*. The *while* structure body may be a single statement or a block. Eventually, the condition becomes false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first statement after the repetition structure executes.

As an example of a **`while`** structure, consider a program segment designed to find the first power of 2 larger than 1000. Suppose **`int`** variable **`product`** contains the value 2. When the following **`while`** structure finishes executing, **`product`** contains the result:

```
int product = 2;

while ( product <= 1000 )
   product = 2 * product;
```

The flowchart in Fig. 4.5 illustrates the flow of control of the preceding **`while`** repetition structure. Once again, note that (besides small circles and arrows) the flowchart contains only a rectangle symbol and a diamond symbol.

Common Programming Error 4.3

Not providing in the body of a **`while`** *structure an action that eventually causes the condition to become false is a logic error. Normally, such a repetition structure will never terminate, which is an error called an “infinite loop.”*

Common Programming Error 4.4

Beginning the keyword **`while`** *with an uppercase* **`W`**, *as in* **`While`**, *is a syntax error. Remember that C# is a case-sensitive language. All of C#’s keywords—***`while`**, **`if`**, **`else`**, *etc.—contain only lowercase letters.*

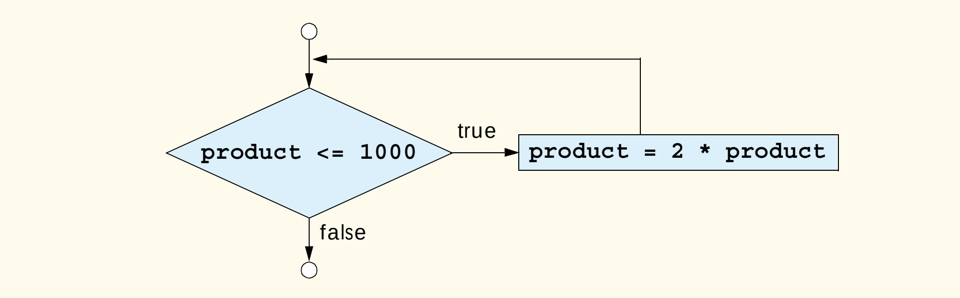


Fig. 4.5 Flowcharting the **`while`** repetition structure.

Testing and Debugging Tip 4.2

Visual Studio .NET will not color a keyword properly unless that keyword is spelled correctly and with the correct case.

Imagine, again, a deep bin of empty **`while`** structures that may be stacked and nested with other control structures to form a structured implementation of an algorithm's flow of control. The empty rectangles and diamonds are filled with appropriate actions and decisions. The flowchart clearly shows the repetition. The flowline emerging from the rectangle indicates that program control continues with the decision, which is tested during each iteration of the loop until the decision eventually becomes false. At this point, the **`while`** structure terminates, and control passes to the next statement following the **`while`** structure in the program.

When the **`while`** structure begins executing, **`product`** is 2. Variable **`product`** is repeatedly multiplied by 2, taking on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024, successively. When **`product`** becomes 1024, the condition **`product <= 1000`** in the **`while`** structure becomes **`false`**. This terminates the repetition with 1024 as **`product`**'s final value. Execution continues with the next statement after the **`while`**. [*Note*: If a **`while`** structure's condition is initially **`false`**, the body statement(s) will never be executed.]

4.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)

To illustrate how algorithms are developed, we solve several variations of a class-averaging problem. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation and display the result.

Let us use pseudocode to list the actions to execute and to specify the order of execution. We use *counter-controlled repetition* to input the grades one at a time. This technique uses a variable called a *counter* to control the number of times a set of statements will execute. In this example, repetition terminates when the counter exceeds 10. This section presents a pseudocode algorithm (Fig. 4.6) and the corresponding program (Fig. 4.7). In

Section 4.9, we show how to develop a pseudocode algorithm. Counter-controlled repetition is also called *definite repetition* because the number of repetitions is known before the loop begins executing.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 4.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```
// Fig. 4.7: Average1.cs
// Class average with counter-controlled repetition.

using System;

class Average1
{
   static void Main( string[] args )
   {
      int total,          // sum of grades
          gradeCounter,   // number of grades entered
          gradeValue,     // grade value
          average;        // average of all grades

      // initialization phase
      total = 0;          // clear total
      gradeCounter = 1;   // prepare to loop

      // processing phase
      while ( gradeCounter <= 10 )  // loop 10 times
      {
         // prompt for input and read grade from user
         Console.Write( "Enter integer grade: " );

         // read input and convert to integer
         gradeValue = Int32.Parse( Console.ReadLine() );

         // add gradeValue to total
         total = total + gradeValue;

```

Fig. 4.7 Class average program with counter-controlled repetition. (Part 1 of 2.)

```
31          // add 1 to gradeCounter
32          gradeCounter = gradeCounter + 1;
33       }
34
35       // termination phase
36       average = total / 10;  // integer division
37
38       // display average of exam grades
39       Console.WriteLine( "\nClass average is {0}", average );
40
41    } // end Main
42
43 } // end class Average1
```

```
Enter integer grade: 100
Enter integer grade: 88
Enter integer grade: 93
Enter integer grade: 55
Enter integer grade: 68
Enter integer grade: 77
Enter integer grade: 83
Enter integer grade: 95
Enter integer grade: 73
Enter integer grade: 62

Class average is 79
```

Fig. 4.7 Class average program with counter-controlled repetition. (Part 2 of 2.)

Note the references in the algorithm (Fig. 4.6) to a total and a counter. The pseudocode variable *total* accumulates the sum of a series of values. A counter is a variable that counts—in this case, that counts the number of grades entered. Variables that store totals normally should be initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

Testing and Debugging Tip 4.3

Initialize counters and totals.

Line 6 begins the definition of class `Average1`. Remember that an application class definition must contain a `Main` method (lines 8–41) to begin execution of the application.

Lines 10–13 declare variables `total`, `gradeCounter`, `gradeValue` and `average` to be of type `int`. Variable `gradeValue` will store the value the user inputs after the value is converted from a `string` to an `int`.

Good Programming Practice 4.4

Always place a blank line between a declaration and executable statements. This makes the declarations stand out in a program and contributes to program clarity.

Lines 16–17 are assignment statements that initialize `total` to `0` and `gradeCounter` to `1`. Variables `total` and `gradeCounter` are initialized before they are used in a calculation. Recall that using uninitialized variables in calculations results in compilation errors.

Line 20 indicates that the `while` structure should continue as long as the value of `gradeCounter` is less than or equal to 10. Lines 23 and 26 correspond to the pseudocode statement *“Input the next grade.”* The statement on line 23 displays the prompt “`Enter integer grade:`” on the screen. The statement on line 26 reads the information entered by the user, converts it to an `int` and stores the value in `gradeValue`. Next, line 29 updates the `total` with the new `gradeValue` by adding `gradeValue` to the previous value of `total` and assigning the result to `total`.

The program is now ready to increment the variable `gradeCounter` to indicate that a grade has been processed. Line 32 adds `1` to `gradeCounter`, so the condition in the `while` structure eventually will become false and terminate the loop. Line 36 assigns the results of the average calculation to variable `average`. Line 39 displays a message containing the string `"Class average is "` followed by the value of variable `average`.

The averaging calculation produces an integer result. Actually, the sum of the grade-point values in this example is 794, which, when divided by 10, yields 79.4. Such numbers with a decimal point are called floating-point numbers; we discuss floating-point numbers in the next section.

4.9 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)

Let us generalize the class-average problem. Consider the following problem:

> *Develop a class-averaging program that processes an arbitrary number of grades each time the program executes.*

In the first class-average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades are to be input. The program must process an arbitrary number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a *sentinel value* (also called a *signal value*, a *dummy value* or a *flag value*) to indicate “end of data entry.” The user inputs all grades and then types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition often is called *indefinite repetition* because the number of repetitions is not known before the loop begins executing.

The sentinel value cannot be confused with an acceptable input value. Grades on a quiz are normally nonnegative integers, thus -1 is an acceptable sentinel value for this problem. A run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and -1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89. The sentinel value, -1, should not enter into the averaging calculation.

Common Programming Error 4.5

Choosing a sentinel value that is also a legitimate data value results in a logic error and may prevent a sentinel-controlled loop from terminating properly, a problem known as an infinite loop.

We approach the class-average program with *top-down, stepwise refinement*, a technique essential to the development of well-structured algorithms. We begin with a pseudocode representation of the *top:*

```
Determine the class average for the quiz
```

The top is a single statement that conveys the overall function of the program. As such, the top is a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail from which to write the C# algorithm. Therefore, we conduct the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they must be performed. This results in the following *first refinement*:

```
Initialize variables
Input, sum up and count the quiz grades
Calculate and print the class average
```

Here, only the sequence structure has been used—the steps listed are to be executed in order, one after the other.

Software Engineering Observation 4.4

Each refinement, including the top, is a complete specification of the algorithm; only the level of detail in each refinement varies.

To proceed to the next level of refinement (i.e., the *second refinement)*, we commit to specific variables. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade and a variable to hold the calculated average. The pseudocode statement

```
Initialize variables
```

may be refined as follows:

```
Initialize total to zero
Initialize counter to zero
```

Notice that only the variables *total* and *counter* are initialized before they are used; the variables *average* and *grade* (for the calculated average and the user input, respectively) need not be initialized because their values are determined as they are calculated or input.

The pseudocode statement

```
Input, sum up and count the quiz grades
```

requires a repetition structure (i.e., a loop) that successively inputs each grade. We do not know how many grades are to be processed, thus we use sentinel-controlled repetition. The user types in legitimate grades one at a time. After the last legitimate grade is typed, the user types the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value. The second refinement of the preceding pseudocode statement is then

```
Input the first grade (possibly the sentinel)

While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)
```

We do not use braces around the pseudocode that forms the body of the *while* structure. We simply indent the pseudocode under the *while* to show that it belongs to the *while* structure.

Note that a value is input both before reaching the loop and at the end of the loop's body. As we enter the loop, the value input before the loop is tested to determine whether it is the sentinel. If so, the loop terminates; otherwise, the body of the loop executes. The body processes the grade, then inputs the next grade. Then, the new grade is tested at the top of the loop to determine if that grade is the sentinel.

The pseudocode statement

```
Calculate and print the class average
```

may be refined as follows:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
Else
    Print "No grades were entered"
```

We test for the possibility of division by zero—a logic error that, if undetected, causes the program to produce invalid output. The complete second refinement of the pseudocode algorithm for the class-average problem is shown in Fig. 4.8.

Testing and Debugging Tip 4.4

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program, possibly printing an error message.

Good Programming Practice 4.5

Include blank lines in pseudocode programs for increased readability. The blank lines separate pseudocode control structures and the program's phases.

```
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)

While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
Else
    Print "No grades were entered"
```

Fig. 4.8 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.

Software Engineering Observation 4.5

Many algorithms can be divided logically into three phases—an initialization phase that initializes the program variables, a processing phase that inputs data values and adjusts program variables accordingly and a termination phase that calculates and prints the results.

The pseudocode algorithm in Fig. 4.8 solves the more general class-averaging problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary.

Software Engineering Observation 4.6

The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to convert the pseudocode to a C# program. Implementing the C# program then normally occurs in a straightforward manner.

The C# program for this pseudocode is shown in Fig. 4.9. Notice from the output that each grade entered is an integer, although the averaging calculation is likely to produce a number with a decimal point. The type **`int`** cannot represent real numbers, so this program uses data type ***`double`*** to handle floating-point numbers.

The program also introduces the *cast operator* (line 44) to handle the type conversion for the averaging calculation. These features are explained in detail in our discussion of Fig. 4.9.

```
 1  // Fig. 4.9: Average2.cs
 2  // Class average with sentinel-controlled repetition.
 3
 4  using System;
 5
 6  class Average2
 7  {
 8     static void Main( string[] args )
 9     {
10        int total,           // sum of grades
11            gradeCounter,    // number of grades entered
12            gradeValue;      // grade value
13
14        double average;      // average of all grades
15
16        // initialization phase
17        total = 0;           // clear total
18        gradeCounter = 0;    // prepare to loop
19
20        // processing phase
21        // prompt for input and convert to integer
22        Console.Write( "Enter Integer Grade, -1 to Quit: " );
23        gradeValue = Int32.Parse( Console.ReadLine() );
24
25        // loop until a -1 is entered by user
26        while ( gradeValue != -1 )
27        {
28           // add gradeValue to total
29           total = total + gradeValue;
```

Fig. 4.9 Class-average program with sentinel-controlled repetition. (Part 1 of 2.)

```
30
31          // add 1 to gradeCounter
32          gradeCounter = gradeCounter + 1;
33
34          // prompt for input and read grade from user
35          // convert grade from string to integer
36          Console.Write( "Enter Integer Grade, -1 to Quit: " );
37          gradeValue = Int32.Parse( Console.ReadLine() );
38
39       } // end while
40
41       // termination phase
42       if ( gradeCounter != 0 )
43       {
44          average = ( double ) total / gradeCounter;
45
46          // display average of exam grades
47          Console.WriteLine( "\nClass average is {0}", average );
48
49       }
50       else
51       {
52          Console.WriteLine( "No grades were entered." );
53       }
54
55    } // end method Main
56
57 } // end class Average2
```

```
Enter Integer Grade, -1 to Quit: 97
Enter Integer Grade, -1 to Quit: 88
Enter Integer Grade, -1 to Quit: 72
Enter Integer Grade, -1 to Quit: -1

Class average is 85.6666666666667
```

Fig. 4.9 Class-average program with sentinel-controlled repetition. (Part 2 of 2.)

In this example, we examine how control structures may be stacked on top of one another, in sequence. The `while` structure (lines 26–39) is followed immediately by an `if` structure (lines 42–51). Much of the code in this program is identical to the code in Fig. 4.7, so we concentrate on the new features in this example.

Line 14 declares variable `average` to be of type `double`. This change allows us to store the result of the class-average calculation as a floating-point number. Line 18 initializes `gradeCounter` to `0` because no grades have been input yet—recall that this program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, variable `gradeCounter` is incremented only when a valid grade value is input.

Notice the differences between sentinel-controlled repetition and the counter-controlled repetition of Fig. 4.7. In counter-controlled repetition, we read a value from the user during each pass of the `while` structure for the specified number of iterations. In sentinel-controlled repetition, we read one value (line 23) before the program reaches the `while` structure. This value is used to determine if the program's flow of control should enter the body of the **while** structure. If the **while** structure condition is false (i.e., the user has entered the sentinel value), the body of the **while** structure does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the value input by the user is processed (added to the **total**). Then, the next value is input from the user before the end of the **while** structure's body. When program control reaches the closing right brace (**}**) of the body (line 39), execution continues with the next test of the **while** structure condition. The new value input by the user determines if the **while** structure's body should execute again. Notice that the next value is input from the user immediately before the **while** structure condition is evaluated (line 37). This allows the program to determine whether the value just input by the user is the sentinel value *before* the program processes that value as a valid grade. If the value is the sentinel value, the **while** structure terminates, and the value is not added to the **total**.

Notice the block that composes the **while** loop in Fig. 4.9. Without the braces, the last three statements in the body of the loop would be outside the loop, causing the computer to interpret the code incorrectly, as follows:

```
while ( gradeValue != -1 )

   // add gradeValue to total
   total = total + gradeValue;

// add 1 to gradeCounter
gradeCounter = gradeCounter + 1;

// prompt for input and read grade from user
Console.Write( "Enter Integer Grade, -1 to Quit: " );
gradeValue = Int32.Parse( Console.ReadLine() );
```

An infinite loop occurs in the program if the user fails to input the sentinel **-1** as the input value at line 23 (before the **while** structure).

Common Programming Error 4.6

Omitting the curly braces that delimit a block in a repetition structure can lead to logic errors, such as infinite loops.

Good Programming Practice 4.6

In a sentinel-controlled loop, the prompts requesting data entry should remind the user of the sentinel value.

Averages do not always evaluate to integer values. Often, an average is a value such as 3.333 or 2.7, that contains a fractional part. These values are floating-point numbers and usually are represented by the data type **double**. We declare the variable **average** as type **double** to capture the fractional result of our calculation. However, the result of the calculation **total / gradeCounter** is an integer because **total** and **gradeCounter** are both integer variables. Dividing two integers results in *integer division*, in which any fractional part of the calculation is *truncated* and the result is a whole number. The calculation is performed first, thus the fractional part is lost before the result is assigned to **average**. To produce a floating-point calculation with integer values, we must create temporary values that are floating-point numbers for the calculation. C# provides the *unary* *cast operator* to create this temporary value. Line 44 uses the cast operator `( double )` to create a temporary floating-point copy of its operand—`total`. Using a cast operator in this manner is called *explicit conversion*. The value stored in `total` is still an integer. The calculation now consists of a floating-point value (the temporary `double` version of `total`) divided by the integer `gradeCounter`. Note that the cast does not modify the value stored in memory for `total`. Rather it creates a temporary value that is used only for this calculation.

Common Programming Error 4.7

Assuming that integer division rounds (rather than truncates) can lead to incorrect results.

C# can evaluate only arithmetic expressions in which the data types of the operands are identical. To ensure that the operands are of the same type, C# performs *implicit conversion* (also called *promotion*) on selected operands. Through implicit conversion, in an expression containing the data types `int` and `double`, `int` operands are *promoted* to `double`. In our example, the temporary `double` version of `total` is divided by the `int` `gradeCounter`. Therefore, a temporary version of `gradeCounter` is promoted to `double`, the calculation is performed and the result of the floating-point division is assigned to `average`.

Cast operators are available for most data types. The cast operator is known as a *unary operator* (i.e., an operator that takes only one operand) and is formed by placing parentheses around a data type name. In Chapter 3, Introduction to C# Programming, we studied the binary arithmetic operators. C# also supports unary versions of the plus (`+`) and minus (`-`) operators, so the programmer can write expressions like `-7` or `+5`. Cast operators associate from right to left and have the same precedence as other unary operators, such as unary `+` and unary `-`. This precedence is one level higher than that of the *multiplicative operators* `*`, `/` and `%` and one level lower than that of parentheses. (See the operator precedence chart in Appendix A.) In our precedence charts, we indicate the cast operator with the notation `(`*type*`)` to show that any type name can form a cast operator.

Common Programming Error 4.8

Using floating-point numbers in a manner that assumes that they are precisely represented real numbers can lead to incorrect results. Real numbers are represented only approximately by computers.

Good Programming Practice 4.7

Do not compare floating-point values for equality or inequality. Rather, test that the absolute value of the difference between two floating-point numbers is less than a specified small value.

Despite the fact that floating-point numbers are not always "100% precise," they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. Calling such a number simply 98.6 is fine for most applications.

Floating-point numbers also develop through division. When we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation.

Line 47 displays the value of **`average`**. We specify average as the second argument to **`WriteLine`**. Method **`WriteLine`** will convert this argument to a **`string`** and display its value.

4.10 Formulating Algorithms with Top-Down, Stepwise Refinement: Case Study 3 (Nested Control Structures)

Let us work through another complete problem. We will again formulate the algorithm using pseudocode and top-down, stepwise refinement; we will write a corresponding C# program.

Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, several of the students who completed this course took the licensing examination. The college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of the 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed the exam.*
>
> *Your program should analyze the results of the exam as follows:*
>
> 1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.*
> 2. *Count the number of test results of each type.*
> 3. *Display a summary of the test results, indicating the number of students who passed and the number of students who failed the exam.*
> 4. *If more than 8 students passed the exam, print the message "Raise tuition."*

After reading the problem statement carefully, we make the following observations about the problem:

1. The program must process test results for 10 students. A counter-controlled loop will be used.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it is a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
3. Two counters keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed.
4. After the program processes all the results, it must decide if more than eight students passed the exam.

Let us proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

```
Analyze exam results and decide if tuition should be raised
```

Once again, it is important to emphasize that the top is a complete representation of the program, but several refinements are likely to be needed before the pseudocode can be evolved naturally into a C# program. Our first refinement is

```
Initialize variables
Input the ten exam grades and count passes and failures
Print a summary of the exam results and decide if tuition should be raised
```

Even though we have a complete representation of the entire program, further refinement is necessary. We must commit to specific variables. Counters are needed to record the passes and failures. A counter controls the looping process and a variable stores the user input. The pseudocode statement

```
Initialize variables
```

may be refined as follows:

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one
```

Only the counters for the number of passes, number of failures and number of students are initialized. The pseudocode statement

```
Input the ten quiz grades and count passes and failures
```

requires a loop that successively inputs the result of each exam. Here, it is known in advance that there are precisely ten exam results, so counter-controlled repetition is appropriate. Inside the loop (i.e., *nested* within the loop) a double-selection structure determines whether each exam result is a pass or a failure, and the structure increments the appropriate counter accordingly. The refinement of the preceding pseudocode statement is

```
While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter
```

Notice the use of blank lines to offset the *If/Else* control structure to improve program readability. The pseudocode statement

```
Print a summary of the exam results and decide if tuition should be raised
```

may be refined as follows:

```
Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Raise tuition"
```

The complete second refinement appears in Fig. 4.10. Notice that blank lines also set off the *While* structure for program readability.

The pseudocode now is refined sufficiently for conversion to C#. The C# program and sample executions are shown in Fig. 4.11.

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print “Raise tuition”
```

Fig. 4.10 Pseudocode for examination-results problem.

```
// Fig. 4.11: Analysis.cs
// Analysis of Examination Results.

using System;

class Analysis
{
   static void Main( string[] args )
   {
      int passes = 0,           // number of passes
          failures = 0,         // number of failures
          student = 1,          // student counter
          result;               // one exam result

      // process 10 students; counter-controlled loop
      while ( student <= 10 )
      {
         Console.Write( "Enter result (1=pass, 2=fail): " );
         result = Int32.Parse( Console.ReadLine() );

         if ( result == 1 )
            passes = passes + 1;

         else
            failures = failures + 1;
```

Fig. 4.11 C# program for examination-results problem. (Part 1 of 2.)

```
26
27           student = student + 1;
28        }
29
30        // termination phase
31        Console.WriteLine();
32        Console.WriteLine( "Passed: " + passes );
33        Console.WriteLine( "Failed: " + failures );
34
35        if ( passes > 8 )
36           Console.WriteLine( "Raise Tuition\n" );
37
38     } // end of method Main
39
40  } // end of class Analysis
```

```
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1

Passed: 9
Failed: 1
Raise Tuition
```

```
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 2
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1
Enter result (1=pass, 2=fail): 1

Passed: 5
Failed: 5
```

Fig. 4.11 C# program for examination-results problem. (Part 2 of 2.)

Lines 10–13 declare the variables used in **`Main`** to process the examination results. We have taken advantage of a C# feature that incorporates variable initialization into declarations (**`passes`** is assigned `0`, **`failures`** is assigned `0` and **`student`** is assigned `1`). Programs that contain repetition may require initialization at the beginning of each repetition;

such initialization normally occurs in assignment statements. Notice the use of the nested **`if`**/**`else`** structure (lines 21–25) in the **`while`** structure's body. Also, notice the new statement at line 31 that uses **`Console.WriteLine`** to output a blank line.

Software Engineering Observation 4.7

The most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C# program from the algorithm is normally straightforward.

Software Engineering Observation 4.8

Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer, and that writing pseudocode merely delays the production of final output. Although this may work for simple and familiar problems, it can lead to serious problems on large, complex projects.

4.11 Assignment Operators

C# provides several assignment operators for abbreviating assignment expressions. For example, the statement

```
c = c + 3;
```

can be abbreviated with the *addition assignment operator* **`+=`** as

```
c += 3;
```

The **`+=`** operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

variable **`=`** *variable operator expression***`;`**

where *operator* is one of the binary operators **`+`**, **`-`**, **`*`**, **`/`** or **`%`** (or others we will discuss later in the book), can be written in the form

*variable operator***`=`** *expression***`;`**

Figure 4.12 includes the arithmetic assignment operators, sample expressions using these operators and explanations.

Common Programming Error 4.9

Placing a space character between symbols that compose an arithmetic assignment operator is a syntax error.

Assignment operator	Sample expression	Explanation	Assigns
Assume: **`int c = 3, d = 5, e = 4, f = 6, g = 12;`**			
`+=`	**`c += 7`**	**`c = c + 7`**	**`10`** to **`c`**

Fig. 4.12 Arithmetic assignment operators. (Part 1 of 2.)

Assignment operator	Sample expression	Explanation	Assigns
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 4.12 Arithmetic assignment operators. (Part 2 of 2.)

4.12 Increment and Decrement Operators

C# provides the unary *increment operator*, `++`, and the unary *decrement operator*, `--`, which are summarized in Fig. 4.13. A program can increment the value of a variable called `c` by 1 using the increment operator, `++`, rather than the expression `c = c + 1` or `c += 1`. If an increment or decrement operator is placed before a variable, it is referred to as the *preincrement* or *predecrement operator*, respectively. If an increment or decrement operator is placed after a variable, it is referred to as the *postincrement* or *postdecrement operator*, respectively.

Preincrementing (or predecrementing) a variable causes the variable to be incremented (or decremented) by 1, and then the new value of the variable is used in the expression in which it appears. Postincrementing (or postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, and then the variable value is incremented (or decremented) by 1.

The application in Fig. 4.14 demonstrates the difference between the preincrementing version and the postincrementing version of the `++` increment operator. Postincrementing the variable `c` causes it to be incremented after it is used in the `Console.WriteLine` method call (line 14). Preincrementing the variable `c` causes it to be incremented before it is used in the `Console.WriteLine` method call (line 21).

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postincrement	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	predecrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postdecrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

Fig. 4.13 The increment and decrement operators.

```
 1  // Fig. 4.14: Increment.cs
 2  // Preincrementing and postincrementing
 3
 4  using System;
 5
 6  class Increment
 7  {
 8     static void Main( string[] args )
 9     {
10        int c;
11
12        c = 5;
13        Console.WriteLine( c );   // print 5
14        Console.WriteLine( c++ ); // print 5 then postincrement
15        Console.WriteLine( c );   // print 6
16
17        Console.WriteLine();      // skip a line
18
19        c = 5;
20        Console.WriteLine( c );   // print 5
21        Console.WriteLine( ++c ); // preincrement then print 6
22        Console.WriteLine( c );   // print 6
23
24     } // end of method Main
25
26  } // end of class Increment
```

```
5
5
6

5
6
6
```

Fig. 4.14 The difference between preincrementing and postincrementing.

The program displays the value of `c` before and after the `++` operator is used. The decrement operator (`--`) works similarly.

Good Programming Practice 4.8

For readability, unary operators should be placed next to their operands, with no intervening spaces.

Line 17,

```
Console.WriteLine();      // skip a line
```

uses `Console.WriteLine` to output a blank line. If `Console.WriteLine` receives no arguments, it simply outputs a newline character.

The arithmetic assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 4.11 (lines 22, 25 and 27)

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with assignment operators as

```
passes += 1;
failures += 1;
student += 1;
```

with preincrement operators as

```
++passes;
++failures;
++student;
```

or with postincrement operators as

```
passes++;
failures++;
student++;
```

It is important to note here that when incrementing or decrementing a variable in an expression or statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect. It is only when a variable appears in the context of a larger expression that preincrementing and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

Common Programming Error 4.10

Attempting to use the increment or decrement operator on an expression other than a variable reference *is a syntax error. A* variable reference *is a variable or expression that can appear on the left side of an assignment operation. For example, writing* `++(x + 1)` *is a syntax error, because* `(x + 1)` *is not a* variable reference.[2]

The chart in Fig. 4.15 shows the precedence and associativity of the operators introduced to this point. The operators are shown top to bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. Notice that the conditional operator (`?:`), the unary operators increment (`++`), decrement (`--`), plus (`+`), minus (`-`), cast and the assignment operators (`=`, `+=`, `-=`, `*=`, `/=` and `%=`) associate from right to left. All other operators in the operator precedence chart of Fig. 4.15 associate from left to right. The third column names the groups of operators.

Operators	Associativity	Type
`()`	left to right	parentheses
`++ --`	right to left	unary postfix

Fig. 4.15 Precedence and associativity of the operators discussed so far in this book. (Part 1 of 2.)

---

[2] The term *variable reference* is equivalent to the term *lvalue* (“left value”), which is popular among C and C++ programmers.

Operators	Associativity	Type
`++ -- + - (`*type*`)`	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 4.15 Precedence and associativity of the operators discussed so far in this book. (Part 2 of 2.)

4.13 Introduction to Windows Application Programming

Today, users demand software with rich GUIs that allow them to click buttons, select items from menus and much more. In this chapter and the previous, we created console applications. However, most C# programs used in industry are Windows applications with GUIs. For this reason, we are introducing Windows applications early in the book, although doing so exposes some concepts that we do not explain fully until later chapters.

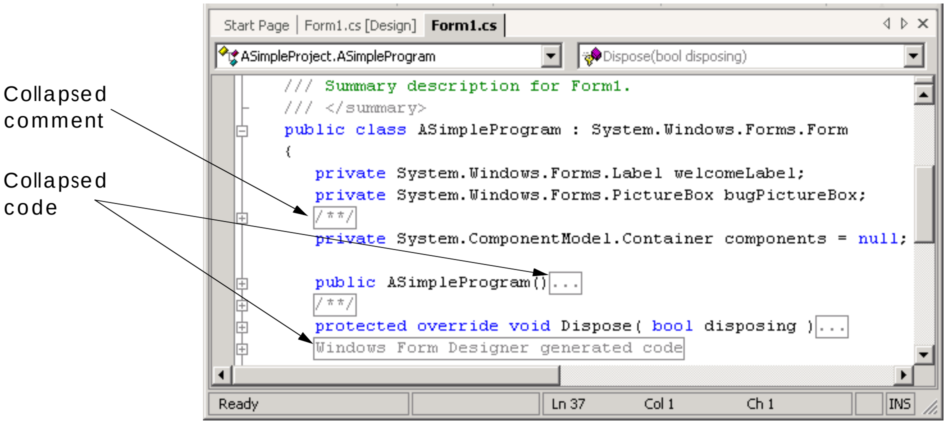
In Chapter 2, Introduction to the Visual Studio .NET IDE, we introduced the concept of visual programming, which allows programmers to create graphical user interfaces (GUIs) without writing any programming code. In this section, we combine visual programming with the conventional programming techniques introduced in this chapter and Chapter 3, Introduction to C# Programming. Through this combination, we can enhance considerably the Windows application introduced in Chapter 2.

Load the project **`ASimpleProject`** from Chapter 2 into the IDE. To identify easily the form and its controls in the program code, change the **`(Name)`** properties of the form, label and picture box to **`ASimpleProgram`**, **`welcomeLabel`** and **`bugPictureBox`**, respectively. To change a GUI component's properties, select (click) the component in the design window, then locate the property in the **Properties** window. Click the box to the right of the property name to input a new value, then press the *Enter* key.

With visual programming, the IDE generates the program code that creates the GUI. This code contains instructions for the creation of the form and every control on it. Unlike a console application, a Windows application's program code is not displayed initially in the editor window. Once the program's project (e.g., **`ASimpleProgram`**) is opened in the IDE, the program code can be viewed by selecting **View > Code**. Figure 4.16 shows the code editor displaying the program code.

Windows applications use classes. We already have seen examples of classes such as **`Console`** and **`MessageBox`**, which are defined within the .NET Framework Class Library. Classes are logical groupings of procedures and data that simplify program organization. In-depth coverage of classes is provided in Chapter 8, Object-Based Programming.

Every Windows application consists of at least one class that *inherits* from class ***`Form`*** (which represents a form) in the .NET Framework Class Library's **`System.Windows.Forms`** namespace. The keyword **`class`** begins a class definition and is followed



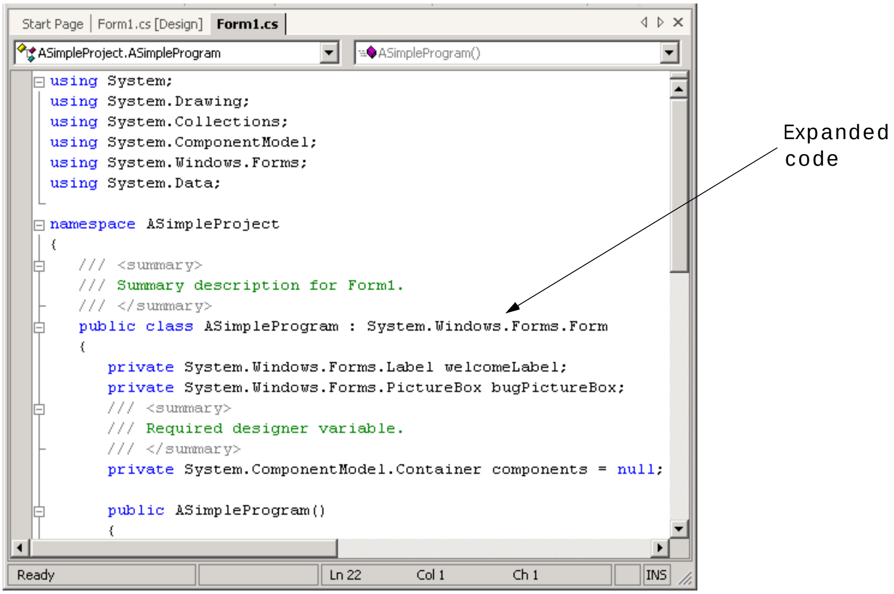
```
/// Summary description for Form1.
/// </summary>
public class ASimpleProgram : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label welcomeLabel;
   private System.Windows.Forms.PictureBox bugPictureBox;
   /**/
   private System.ComponentModel.Container components = null;

   public ASimpleProgram()...
   /**/
   protected override void Dispose( bool disposing )...
   Windows Form Designer generated code
```

Fig. 4.16 IDE showing program code for Fig. 2.15.

immediately by the class name (**`ASimpleProgram`**). Recall that the form's name is set using the **`(Name)`** property. A colon (**`:`**) indicates that the class **`ASimpleProgram`** inherits existing pieces from another class. The class from which **`ASimpleProgram`** inherits—here, **`System.Windows.Forms.Form`**—appears to the right of the colon. In this inheritance relationship, **`Form`** is called the *base class* (or *superclass*), and **`ASimpleProgram`** is called the *derived class* (or *subclass*). With inheritance **`ASimpleProgram`**'s class definition has the *attributes* (data) and *behaviors* (methods) of class **`Form`**. We discuss the significance of the keyword **`public`** in Chapter 6. [*Note*: Changing a control's name in the **Properties** window may not change all occurrences of the control's name in the code. The reader should search the code and replace names that were not changed by the IDE. For example, the original form name (and class name) was **`Form1`**. Search the code for **`Form1`** and change any remaining instances to **`ASimpleProgram`**.]

A key benefit of inheriting from class **`Form`** is that someone else has previously defined "what it means to be a form." The Windows operating system expects every window (e.g., form) to have certain attributes and behaviors. However, because class **`Form`** already provides those capabilities, programmers do not need to "reinvent the wheel" by defining all those capabilities themselves. In fact, class **`Form`** has over 400 methods! In our programs up to this point, we have used only one method (i.e., **`Main`**), so you can imagine how much work went into creating class **`Form`**. The use of the colon to extend from class **`Form`** enables programmers to create forms quickly.

In the editor window (Fig. 4.16), notice the text **`Windows Form Designer generated code`**, which is colored gray and has a plus box next to it. The plus box indicates that this section of code is *collapsed*. Although collapsed code is not visible, it is still part of the program. Code collapsing allows programmers to hide code in the editor, so that they can focus on smaller (and perhaps more important) code segments. Notice that the entire class definition also can be collapsed by clicking the minus box to the left of **`public`**. In Fig. 4.16, the description in gray indicates that the collapsed code was created by the *Windows Form Designer* (i.e., the part of the IDE that creates the code for the GUI). This collapsed code contains the code created by the IDE for the form and its controls, as well as code that enables the program to run. Click the plus box to view the code.

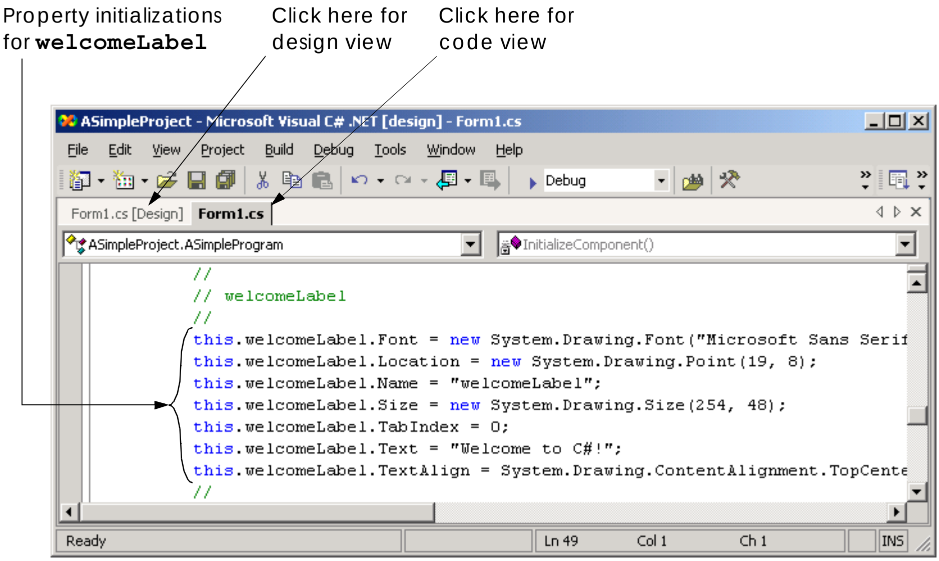


```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ASimpleProject
{
   /// <summary>
   /// Summary description for Form1.
   /// </summary>
   public class ASimpleProgram : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Label welcomeLabel;
      private System.Windows.Forms.PictureBox bugPictureBox;
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components = null;

      public ASimpleProgram()
      {
```

Fig. 4.17 Windows Form Designer generated code when expanded.

Upon initial inspection, the *expanded code* (Fig. 4.17) looks incredibly complex.This code is created by the IDE and normally is not edited by the programmer. We feel it is important for novice programmers to see the amount of code that is generated by the IDE, even though much of the code is not explained until later in the book. This type of code is present in every Windows application. Allowing the IDE to create this code saves the programmer considerable development time. If the IDE did not provide the code, the programmer would have to write it, and this would require a considerable amount of time. The vast majority of the code shown has not been introduced yet, so you are not expected to understand how it works. However, certain programming constructs, such as comments and control structures, should be familiar. Our explanation of this code will enable us to discuss visual programming in greater detail. As you continue to study C#, especially in Chapters 8–13, the purpose of this code will become clearer.

When we created this application in Chapter 2, we used the **Properties** window to set properties for the form, label and picture box. Once a property was set, the form or control was updated immediately. Forms and controls contain a set of *default properties*, which are displayed initially in the **Properties** window when a form or control is selected. These default properties provide the initial characteristics of a form or control when it is created. When a control, such as a label, is placed on the form, the IDE adds code to the class (e.g., **`ASimpleProgram`**) that creates the control and that sets some of the control's property values, such as the name of the control and its location on the form. Figure 4.18 shows a portion of the code generated by the IDE for setting the label's (i.e., **`welcomeLabel`**'s) properties. These include the label's **`Font`**, **`Location`**, **`Name`**, **`Text`** and **`TextAlign`** properties. Recall from Chapter 2 that we explicitly set values for the label's **`Name`**, **`Text`** and **`TextAlign`** properties. Other properties, such as **`Location`** are set only when the label is placed on the form.

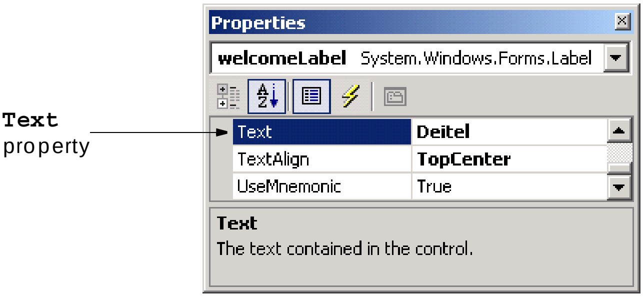

Property initializations for **welcomeLabel**
Click here for design view
Click here for code view
ASimpleProject - Microsoft Visual C# .NET [design] - Form1.cs
File Edit View Project Build Debug Tools Window Help
Debug
Form1.cs [Design]
Form1.cs
ASimpleProject.ASimpleProgram
InitializeComponent()
Ready
Ln 49
Col 1
Ch 1
INS

```
//
// welcomeLabel
//
this.welcomeLabel.Font = new System.Drawing.Font("Microsoft Sans Serif
this.welcomeLabel.Location = new System.Drawing.Point(19, 8);
this.welcomeLabel.Name = "welcomeLabel";
this.welcomeLabel.Size = new System.Drawing.Size(254, 48);
this.welcomeLabel.TabIndex = 0;
this.welcomeLabel.Text = "Welcome to C#!";
this.welcomeLabel.TextAlign = System.Drawing.ContentAlignment.TopCente
//
```

Fig. 4.18 Code generated by the IDE for `welcomeLabel`.

The values assigned to the properties are based on the values in the **Properties** window. We now demonstrate how the IDE updates the Windows Form Designer generated code it generates when a property value in the **Properties** window changes. During this process, we must switch between code view and design view. To switch views, select the corresponding tabs—**Form1.cs*** for code view and **Form1.cs* [Design]** for design view. Alternatively, the programmer can select **View > Code** or **View > Designer**. Perform the following steps:

1. *Modify the label control's* **`Text`** *property using the* ***Properties*** *window*. Recall that properties can be changed in design view by clicking a form or control to select it, then modifying the appropriate property in the **Properties** window. Change the `Text` property of the label to "`Deitel`" (Fig. 4.19).

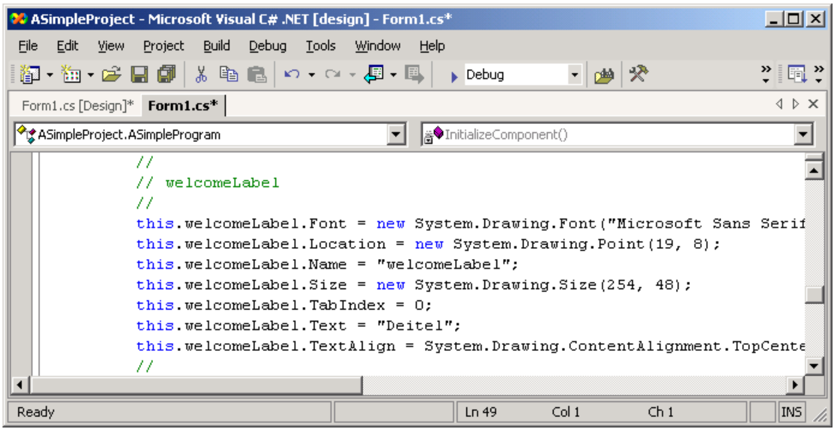

Text property
Properties
welcomeLabel System.Windows.Forms.Label

Text	Deitel
TextAlign	TopCenter
UseMnemonic	True

Text
The text contained in the control.

Fig. 4.19 Using the **Properties** window to set a property value.

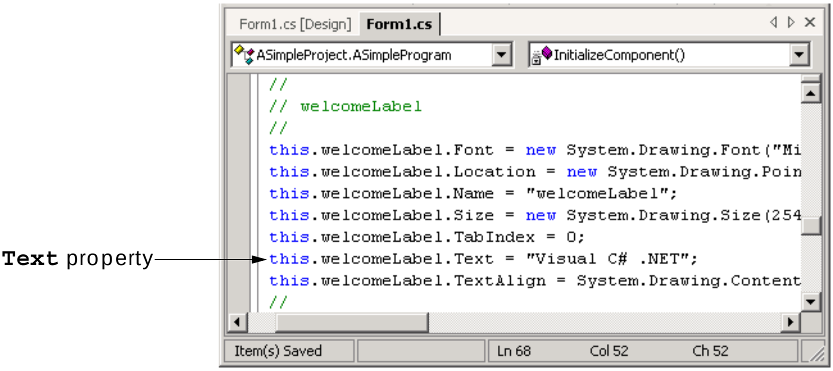
2. *Examine the changes in the code view*. Switch to code view and examine the code. Notice that the label's **`Text`** property is now assigned the text that we entered in the **Properties** window (Fig. 4.20). When a property is changed in design mode, the Windows Form Designer updates the appropriate line of code in the class to reflect the new value.
3. *Modifying a property value in code view*. In the code view editor, locate the three lines of comments indicating the initialization for **`welcomeLabel`** and change the **`string`** assigned to **`this.welcomeLabel.Text`** from "**`Deitel`**" to "**`Visual C# .NET`**" (Fig. 4.21). Now, switch to design mode. The label now displays the updated text, and the **Properties** window for **`welcomeLabel`** displays the new **`Text`** value (Fig. 4.22). [*Note*: Property values should not be set using the techniques presented in this step. Here, we modify the property value in the IDE generated code only as a demonstration of the relationship between program code and the Windows Form Designer.]



ASimpleProject - Microsoft Visual C# .NET [design] - Form1.cs*
File Edit View Project Build Debug Tools Window Help
Debug
Form1.cs [Design]* Form1.cs*
ASimpleProject.ASimpleProgram
InitializeComponent()
Ready
Ln 49 Col 1 Ch 1 INS

```
//
// welcomeLabel
//
this.welcomeLabel.Font = new System.Drawing.Font("Microsoft Sans Serif
this.welcomeLabel.Location = new System.Drawing.Point(19, 8);
this.welcomeLabel.Name = "welcomeLabel";
this.welcomeLabel.Size = new System.Drawing.Size(254, 48);
this.welcomeLabel.TabIndex = 0;
this.welcomeLabel.Text = "Deitel";
this.welcomeLabel.TextAlign = System.Drawing.ContentAlignment.TopCente
//
```

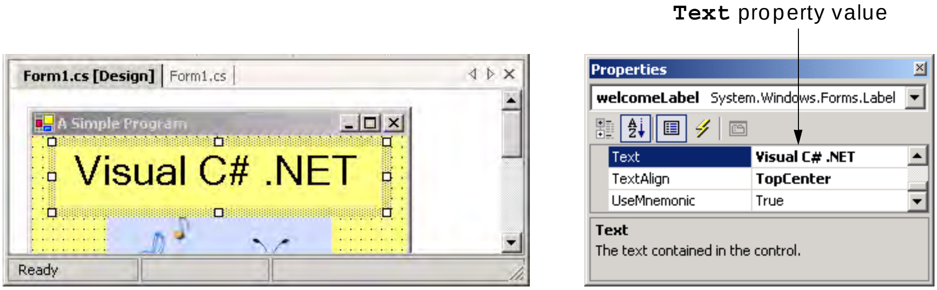
Fig. 4.20 Windows Form Designer generated code reflecting new property values.



Form1.cs [Design] Form1.cs
ASimpleProject.ASimpleProgram
InitializeComponent()
Text property
Item(s) Saved
Ln 68 Col 52 Ch 52

```
//
// welcomeLabel
//
this.welcomeLabel.Font = new System.Drawing.Font("Mi
this.welcomeLabel.Location = new System.Drawing.Poin
this.welcomeLabel.Name = "welcomeLabel";
this.welcomeLabel.Size = new System.Drawing.Size(254
this.welcomeLabel.TabIndex = 0;
this.welcomeLabel.Text = "Visual C# .NET";
this.welcomeLabel.TextAlign = System.Drawing.Content
//
```

Fig. 4.21 Changing a property in the code view editor.

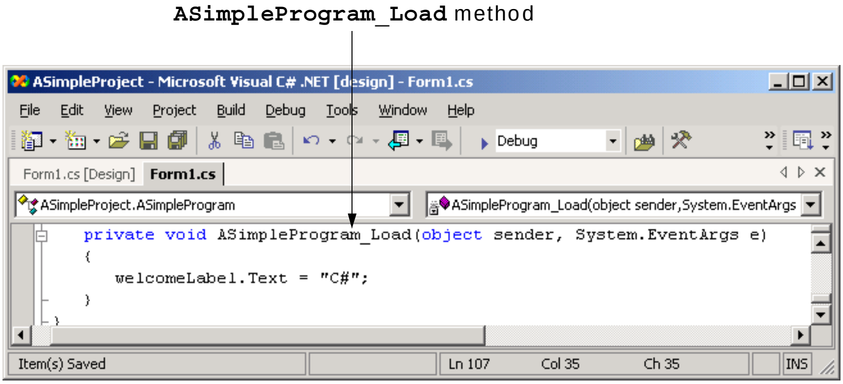

Text property value
Form1.cs [Design]
Form1.cs
A Simple Program
Visual C# .NET
Ready
Properties
welcomeLabel System.Windows.Forms.Label

Text: Visual C# .NET
TextAlign: TopCenter
UseMnemonic: True

Text
The text contained in the control.

Fig. 4.22 New `Text` property value reflected in design mode.

4. *Change the label's* ***`Text`*** *property at runtime.* In the previous steps, we set properties at design time. Often, however, it is necessary to modify a property while a program is running. For example, to display the result of a calculation, a label's text can be assigned a **`string`** containing the result. In console applications, such code is located in **`Main`**. In Windows applications, we must create a method that executes when the form is loaded into memory during program execution. Like **`Main`**, this method is invoked when the program is run. Double-clicking the form in design view adds a method named ***`ASimpleProgram_Load`*** to the class (Fig. 4.23). The cursor is placed in the body of the **`ASimpleProgram_Load`** method definition. Notice that **`ASimpleProgram_Load`** is not part of the Windows Form Designer generated code. Add the statement **`welcomeLabel.Text = "C#";`** in the body of the method definition (Fig. 4.23). In C#, properties are accessed by placing the property name (i.e., **`Text`**) after the object name (i.e., **`welcomeLabel`**), separated by the dot operator. This syntax is similar to that used when accessing object methods. Notice that the *IntelliSense* feature displays the **`Text`** property in the member list after the class name and dot operator have been typed (Fig. 4.29). In Chapter 8, Object-Based Programming, we discuss how programmers can create their own properties.


ASimpleProgram_Load method
ASimpleProject - Microsoft Visual C# .NET [design] - Form1.cs
File Edit View Project Build Debug Tools Window Help
Debug
Form1.cs [Design]
Form1.cs
ASimpleProject.ASimpleProgram
ASimpleProgram_Load(object sender,System.EventArgs
Item(s) Saved
Ln 107
Col 35
Ch 35
INS

```
private void ASimpleProgram_Load(object sender, System.EventArgs e)
{
   welcomeLabel.Text = "C#";
}
}
```

Fig. 4.23 Method `FrmASimpleProgram_Load`.

5. *Examine the results of the* `ASimpleProgram_Load` *method.* Notice that the text in the label looks the same in **Design** mode as it did in Fig. 4.22. Note also that the property window still displays the value "`Visual C# .NET`" as the label's `Text` property and that the IDE generated code has not changed either. Select **Build > Build** then **Debug > Start** to run the program. Once the form is displayed, the text in the label reflects the property assignment in `ASimpleProgram_Load` (Fig. 4.24).
6. *Terminate program execution.* Click the close button to terminate program execution. Once again, notice that both the label and the label's `Text` property contain the text `Visual C# .NET`. The IDE generated code also contains the text `Visual C# .NET`, which is assigned to the label's `Text` property.

In this chapter, we introduced program building blocks called control structures. We also discussed aspects of Windows application programming. In Chapter 5, Control Structures: Part 2, we continue our discussion of control structures by presenting additional selection and repetition structures. In addition, we also build upon the Windows application concepts presented in this chapter by creating a richer Windows application.

SUMMARY

- Executing a series of actions in a specific order can solve many computing problems.
- A procedure for solving a problem in terms of the actions to execute and the order in which these actions execute is an algorithm.
- Program control specifies the order in which statements execute in a computer program.
- Pseudocode is an artificial and informal language that helps programmers develop algorithms and "think out" a program during the program design process.
- C# code corresponds closely to pseudocode. This is a property of pseudocode that makes it a useful program development tool.
- Normally, statements in a program execute one after the other in the order in which they appear. This is called sequential execution.

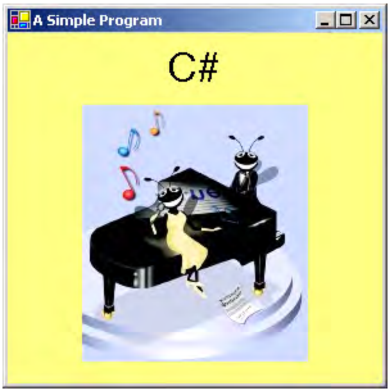


Fig. 4.24 Changing a property value at runtime.

- Various C# statements enable the programmer to specify that the next statement to execute may be other than the next one in sequence. This is called transfer of control.
- Many programming complications in the 1960's were a result of misusing the **`goto`** statement, which allows the programmer to specify a transfer of control to one of a wide range of possible destinations in a program. The notion of structured programming became almost synonymous with "**`goto`** elimination."
- Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures—namely, sequence, selection and repetition.
- The sequence structure is built into C#. Unless directed otherwise, the computer executes C# statements one after the other in the order in which they appear.
- A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called flowlines, which indicate the order in which the algorithm's actions execute.
- The **`if`** selection structure performs (selects) an action if a condition is true or skips the action if the condition is false.
- The **`if`**/**`else`** selection structure performs an action if a condition is **`true`** and performs a different action if the condition is **`false`**.
- A single-selection structure is one that selects or ignores a single action.
- A double-selection structure is one that selects between two actions.
- A multiple-selection structure is one that selects among many actions.
- Keywords are reserved by the language to implement various features, such as C#'s control structures. Keywords cannot be used as identifiers.
- Each program is formed by combining as many of each type of C#'s eight control structures as is appropriate for the algorithm the program implements.
- Single-entry/single-exit control structures make it easy to build programs. The control structures are attached to one another by connecting the exit point of one control structure to the entry point of the next. This is called control-structure stacking.
- Algorithms in C# programs are constructed from only eight different types of control structures combined in only two ways.
- The decision symbol has two flowlines emerging from it. One indicates the direction to be taken when the expression in the symbol is true; the other indicates the direction to be taken when the expression is false.
- Control structure flowcharts contain (besides small circle symbols and flowlines) only rectangle symbols to indicate the actions to be performed and diamond symbols to indicate decisions to be made. This is the action/decision model of programming.
- The ternary conditional operator (**`?:`**) is closely related to the **`if`**/**`else`** structure. The operands and the **`?:`** form a conditional expression. The first operand is a condition that evaluates to a **`bool`** value, the second is the value for the conditional expression if the condition evaluates to **`true`** and the third is the value for the conditional expression if the condition evaluates to **`false`**.
- Nested **`if`**/**`else`** structures test for multiple cases by placing **`if`**/**`else`** structures inside other **`if`**/**`else`** structures.
- A set of statements in a pair of braces is called a block. A block can be placed anywhere in a program that a single statement can be placed.
- A syntax error is caught by the compiler at compile time, while a logic error has its effect during execution.

- A fatal logic error causes a program to fail and terminate prematurely. A nonfatal logic error allows a program to continue executing, but the program produces incorrect results.
- A repetition structure repeats an action (or set of actions) while some condition remains true.
- Eventually, the condition in a **`while`** structure will become false. At this point, the repetition terminates, and the first statement after the repetition structure executes.
- It is a logic error to fail to provide in the body of a **`while`** structure an action that eventually causes the condition to become false. Normally, such a repetition structure will never terminate, which is an error called an "infinite loop."
- We use counter-controlled repetition to input data values one at a time, a specified number of times. This technique uses a variable called a *counter* to control the number of times a set of statements will execute.
- Counter-controlled repetition often is called definite repetition because the number of repetitions is known before the loop begins executing.
- Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing.
- The sentinel value (also called the signal value, dummy value or flag value) determines when to terminate a repetition structure.
- We approach programming problems with top-down, stepwise refinement—a technique that is essential to the development of well-structured algorithms.
- The top is a single statement that conveys the overall function of the program. As such, the top is a complete representation of a program.
- We divide the top into a series of smaller tasks and list these in the order in which they must be performed. Each refinement, including the top itself, is a complete specification of the algorithm; only the level of detail in each refinement varies.
- Many algorithms can be divided logically into three phases—an initialization phase that initializes the program variables, a processing phase that inputs data values and adjusts program variables accordingly and a termination phase that calculates and prints the results.
- The programmer terminates the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for the programmer to convert the pseudocode to a C# program.
- Omitting the curly braces that delineate a block in the body of a repetition structure can lead to logic errors, such as infinite loops.
- Dividing two integers results in integer division, in which any fractional part of the calculation is truncated.
- To ensure that the operands in an expression are of the same type, C# performs implicit conversion on selected operands and promotes them to the same type.
- C# provides the unary increment operator, **`++`**, and the unary decrement operator, **`--`**. These operators add 1 to or subtract 1 from their operand, respectively.
- If an increment or decrement operator is placed before a variable, it is referred to as the preincrement or predecrement operator, respectively.
- If an increment or decrement operator is placed after a variable, it is referred to as the postincrement or postdecrement operator, respectively.
- A key benefit of extending classes using inheritance is that all the general capabilities are provided by the original class—programmers do not need to define these capabilities on their own.
- Method **`InitializeComponent`** contains the code to configure component properties in a GUI.
- The value in parentheses after the type in a **`new`** operation initializes the new object.

- Visual Studio .NET generates code that builds the GUI for an application.
- The primitive types are the building blocks for more complicated types.

TERMINOLOGY

`--`, unary decrement operator
`%=`, modulus assignment operator
`(`*type*`)`, cast operator
`*=`, multiplication assignment operator
`++`, unary increment operator
`+=`, addition assignment operator
`/=`, division assignment operator
`;`, empty statement
`=`, assignment operator
`-=`, subtraction assignment operator
`?:`, ternary conditional operator
`{`, open brace
`}`, close brace
abbreviating an assignment expression
action symbol
action/decision model of programming
algorithm
application class definition
assignment operator (`=`)
associate left to right
associate right to left
associativity of operators
binary arithmetic operator
block
body of the **`while`**
`bool` primitive data type
boolean expression
braces that delimit a block
building block
case-sensitive language
cast operator
collapsed code
complete representation of a program
conditional expression
conditional operator (`?:`)
connector symbol
control structure
control-structure nesting
control-structure stacking
counter
counter-controlled repetition
dangling-else problem
decision symbol
declaration
definite repetition
design phase
diamond symbol
`Dispose` method
division by zero
`do`/**`while`** repetition structure
`double` primitive data type
double-selection structure
`else` statement
empty statement (`;`)
end of data entry
`#endregion` directive
entry point of control structure
examination-results problem
exit point of control structure
expanded code
explicit conversion
`false`
fatal logic error
first refinement
flag value
floating-point data type
floating-point division
floating-point number
flow of control
flowchart
flowline
`for` repetition structure
fractional result
`goto` elimination
“**`goto`**-less programming”
graphical representation of an algorithm
`if` selection structure
`if`/**`else`** selection structure
implicit conversion
indefinite repetition
indentation
indentation convention
infinite loop
inheriting from
    `System.Windows.Forms.Form` class
initialization phase
initialize
`InitializeComponent` method
IntelliSense
input/output operation
integer division
integral data type

keyword
level of refinement
logic error
loop
main form
multiple-selection structure
multiplicative operators: `*`, `/` and `%`
nonfatal logic error
oval symbol
postdecrement operator
postdecrementing
postincrement operator
postincrementing
precedence of operators
predecrement operator
predecrementing
preincrement operator
preincrementing
preprocessor directives
primitive (or built-in) data type
procedure for solving a problem
processing phase
program control
program development tool
promotion
pseudocode
pseudocode algorithm
real number
rectangle symbol
refinement process
`#region` directive
repetition structure
second refinement
selection structure
sentinel-controlled repetition
sentinel value
sequence structure
sequential execution
signal value
single-entry/single-exit control structure
single-selection structure
small circle symbol
`string` primitive data type
strongly typed language
structured programming
`switch` selection structure
syntax error
`System.Windows.Forms.Form` class
temporary value
termination phase
ternary operator (`?:`)
top-down, stepwise refinement
transfer of control
`true`
truncate
unary operator
Unicode
variable reference
vertical spacing
`while` repetition structure
white-space characters

SELF-REVIEW EXERCISES

4.1 Fill in the blanks in each of the following statements:

a) All programs can be written in terms of three types of control structures: __________, __________ and __________.

b) The __________ selection structure executes one action when a condition is true and another action when a condition is false.

c) Repetition of a set of instructions a specific number of times is called __________ repetition.

d) When it is not known in advance how many times a set of statements will be repeated, a __________ value can be used to terminate the repetition.

e) Specifying the order in which statements are to be executed in a computer program is called __________.

f) __________ is an artificial and informal language that helps programmers develop algorithms.

g) __________ are reserved by C# to implement various features, such as the language's control structures.

h) A(n) __________ statement specifying that no action is to be taken is indicated by placing a semicolon where a statement normally would be.

i) The increment operator (`++`) and decrement operator (`--`) increment and decrement a variable's value by __________.
j) Explicit conversion makes use of the __________ operator.

4.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) It is difficult to convert pseudocode into a working C# program.
b) Sequential execution refers to statements in a program that execute one after another.
c) It is recommended for C# programmers to use `goto` statements.
d) The `if` structure is called a single-selection structure.
e) Structured programs are clear, easy to debug and modify and more likely than unstructured programs to be bug-free in the first place.
f) The sequence structure is not built into C#.
g) Pseudocode usually resembles actual C# code.
h) Placing a semicolon after the condition in an `if` structure is a syntax error.
i) The `while` structure body may be a single or a block.

4.3 Write four different C# statements that each add 1 to integer variable `x` and store the result in `x`.

4.4 Write C# statements to accomplish each of the following:
a) Assign the sum of `x` and `y` to `z` then increment `x` by 1 after the calculation. Use only one statement.
b) Test if the value of the variable `count` is greater than 10. If it is, print `"Count is greater than 10"`.
c) Decrement the variable `x` by 1, then subtract it from the variable `total`. Use only one statement.
d) Calculate the remainder after `q` is divided by `divisor` and assign the result to `q`. Write this statement two different ways.

4.5 Write a C# statement to accomplish each of the following tasks:
a) Declare variables `sum` and `x` to be of type `int`.
b) Assign `1` to variable `x`.
c) Assign `0` to variable `sum`.
d) Add variable `x` to variable `sum` and assign the result to variable `sum`.
e) Print `"The sum is : "` followed by the value of variable `sum`.

4.6 Combine the statements that you wrote in Exercise 4.5 into a C# application that calculates and prints the sum of the integers from 1 to 10. Use the `while` structure to loop through the calculation and increment statements. The loop should terminate when the value of `x` becomes 11.

4.7 Determine the values of each variable after the calculation is performed. Assume that when each statement begins executing, all variables have the integer value 5.

a)
```
product *= x++;
```
b)
```
quotient /= ++x;
```

4.8 Identify and correct the errors in each of the following:

a)
```
while ( c <= 5 )
{
   product *= c;
   ++c;
```
b)
```
if ( gender == 1 )
   Console.WriteLine( "Woman" );
else;
   Console.WriteLine( "Man" );
```

4.9 What is wrong with the following `while` repetition structure?

```
while ( z >= 0 )
   sum += z;
```

ANSWERS TO SELF-REVIEW EXERCISES

4.1 a) sequence, selection, repetition. b) `if`/`else`. c) counter-controlled or definite. d) sentinel, signal, flag or dummy. e) program control. f) Pseudocode. g) Keywords. h) empty. i) one. j) cast.

4.2 a) False. Pseudocode should convert easily into C# code. b) True. c) False. Some programmers argue that `goto` statements violate structured programming and cause considerable problems. d) True. e) True. f) False. The sequence structure is built into C#; lines of code execute in the order in which they are written, unless explicitly directed to do otherwise. g) True. h) False. Placing a semicolon after the condition in an `if` structure is usually a logic error. i) True.

4.3

```
x = x + 1;
x += 1;
++x;
x++;
```

4.4 a) `z = x++ + y;`

b)
```
if ( count > 10 )
   Console.WriteLine( "Count is greater than 10" );
```

c) `total -= --x;`

d) `q %= divisor;`
`q = q % divisor;`

4.5 a) `int sum, x;`

b) `x = 1;`

c) `sum = 0;`

d) `sum += x;` or `sum = sum + x;`

e) `Console.WriteLine( "The sum is: " + sum );` or
`Console.WriteLine( "The sum is: {0}", sum );`

4.6

```
// Calculate the sum of the integers from 1 to 10

using System;

class Calculate
{
   static void Main( string[] args )
   {
      int sum, x;

      x = 1;
      sum = 0;

      while ( x <= 10 )
      {
         sum += x++;
      }
```

```

         Console.WriteLine( "The sum is: " + sum );
      }
   }
```

4.7 a) `product = 25, x = 6;`
b) `quotient = 0, x = 6;`

4.8 a) Error: Missing the closing right brace of the `while` body.
Correction: Add closing right brace after the statement `++c;`.
b) Error: Semicolon after `else` results in a logic error. The second output statement will always be executed.
Correction: Remove the semicolon after `else`.

4.9 The value of the variable `z` is never changed in the `while` structure. Therefore, if the loop-continuation condition `( z >= 0 )` is true, an infinite loop is created. To prevent the infinite loop, `z` must be decremented so that it eventually becomes less than 0.

EXERCISES

4.10 Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a C# program that will input the miles driven and gallons used (both as doubles) for each tankful. The program should calculate and display the miles per gallon obtained for each tankful and print the combined miles per gallon obtained for all tankfuls up to this point. All average calculations should produce floating-point results.

4.11 Develop a C# application that will determine if a department store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a) Account number
b) Balance at the beginning of the month
c) Total of all items charged by this customer this month
d) Total of all credits applied to this customer's account this month
e) Allowed credit limit

The program should input as integers each of these facts, calculate the new balance (= *beginning balance* + *charges* – *credits*), display the new balance and determine if the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the message, "Credit limit exceeded."

4.12 Write a C# application that uses looping to print the following table of values:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

4.13 *(Dangling-Else Problem)* Determine the output for each of the following, when `x` is `9` and `y` is `11` and when `x` is `11` and `y` is `9`. Note that the compiler ignores the indentation in a C# program. Also, the C# compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces (`{}`). On first glance, the programmer may not be sure which `if` and `else` match; this is referred to as the “dangling-else” problem. We have eliminated the indentation from the following code to make the problem more challenging. (*Hint*: Apply indentation conventions that you have learned.)

a)

```
if ( x < 10 )
if ( y > 10 )
Console.WriteLine( "*****" );
else
Console.WriteLine( "#####" );
Console.WriteLine( "$$$$$" );
```

b)

```
if ( x < 10 ) {
if ( y > 10 )
Console.WriteLine( "*****" );
}
else {
Console.WriteLine( "#####" );
Console.WriteLine( "$$$$$" );
}
```

4.14 A palindrome is a number or a text phrase that reads the same backwards as forwards. For example, each of the following five-digit integers are palindromes: 12321, 55555, 45554 and 11611. Write an application that reads in a five-digit integer and determines whether it is a palindrome. If the number is not five digits, display an error message dialog indicating the problem to the user. When the user dismisses the error dialog, allow the user to enter a new value.

4.15 A company wants to transmit data over the telephone, but they are concerned that their phones may be tapped. All their data are transmitted as four-digit integers. They have asked you to write a program that will encrypt their data so that it may be transmitted more securely. Your application should read a four-digit integer entered by the user in an input dialog and encrypt it as follows: Replace each digit by *(the sum of that digit plus 7) modulus 10*. Then swap the first digit with the third, and swap the second digit with the fourth. Print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it to form the original number.

4.16 The factorial of a nonnegative integer n is written $n!$ (pronounced “n factorial”) and is defined as follows:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1 \quad \text{(for values of } n \text{ greater than or equal to 1)}$$

and

$$n! = 1 \quad (\text{for } n = 0).$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120.

a) Write an application that reads a nonnegative integer from an input dialog and computes and prints its factorial.

b) Write an application that estimates the value of the mathematical constant e by using the formula

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Write an application that computes the value of e^x by using the formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

5

Control Structures: Part 2

Objectives

- To be able to use the **`for`** and **`do`**/**`while`** repetition structures to execute statements in a program repeatedly.
- To understand multiple selection that uses the **`switch`** selection structure.
- To be able to use the **`break`** and **`continue`** program-control statements.
- To be able to use the logical operators.

Who can control his fate?
William Shakespeare, *Othello*

The used key is always bright.
Benjamin Franklin

Man is a tool-making animal.
Benjamin Franklin

Intelligence … is the faculty of making artificial objects, especially tools to make tools.
Henri Bergson

Outline

5.1 Introduction
5.2 Essentials of Counter-Controlled Repetition
5.3 `for` Repetition Structure
5.4 Examples Using the `for` Structure
5.5 `switch` Multiple-Selection Structure
5.6 `do`/`while` Repetition Structure
5.7 Statements `break` and `continue`
5.8 Logical and Conditional Operators
5.9 Structured-Programming Summary

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

5.1 Introduction

Chapter 4 began our introduction to the types of building blocks that are available for problem solving and used those building blocks to implement proven program-construction principles. In this chapter, we continue our presentation of the theory and principles of structured programming by introducing C#'s remaining control structures. As in Chapter 4, the C# techniques you learn here are applicable to most high-level languages. When we begin our formal treatment of object-based programming in C# in Chapter 8, we will see that the control structures we study in this chapter and in Chapter 4 are helpful in building and manipulating objects.

5.2 Essentials of Counter-Controlled Repetition

In the last chapter, we introduced the concept of counter-controlled repetition. In this section, we formalize the elements needed in counter-controlled repetition, namely:

1. The *name* of a *control variable* (or loop counter), used to determine whether the loop continues.
2. The *initial value* of the control variable.
3. The *increment* (or *decrement*) by which the control variable is modified each time through the loop (also known as *each iteration of the loop*).
4. The condition that tests for the *final value* of the control variable (i.e., whether looping should continue).

To see the four elements of counter-controlled repetition, consider the simple program in Fig. 5.1, which displays the digits 1–5.

The declaration (line 10)

```
int counter = 1;
```

names the control variable (`counter`), declares it to be an integer, reserves space for it in memory and sets it to an *initial value* of `1`. This statement is a declaration that includes an

```
 1  // Fig. 5.1: WhileCounter.cs
 2  // Counter-controlled repetition.
 3
 4  using System;
 5
 6  class WhileCounter
 7  {
 8     static void Main( string[] args )
 9     {
10        int counter = 1;          // initialization
11
12        while ( counter <= 5 )    // repetition condition
13        {
14           Console.WriteLine( counter );
15           counter++;             // increment
16
17        } // end while
18
19     } // end method Main
20
21  } // end class WhileCounter
```

```
1
2
3
4
5
```

Fig. 5.1 Counter-controlled repetition with **`while`** structure.

initialization. The declaration and initialization of **`counter`** could also have been accomplished with the declaration and statement

```
int counter;   // declare counter
counter = 1;   // initialize counter to 1
```

The declaration is not executable, but the assignment statement is. We use both approaches to initialization throughout this book.

Lines 12–17 define the **`while`** structure. During each iteration of the loop, line 14 displays the current value of **`counter`**, and line 15 *increments* the control variable by **`1`** upon each iteration of the loop. The loop-continuation condition in the **`while`** structure tests whether the value of the control variable is less than or equal to **`5`** (the *final value* for which the condition is true). The body of this **`while`** is performed even when the control variable is **`5`**. The loop terminates when the control variable exceeds **`5`** (i.e., **`counter`** becomes **`6`**).

The program in Fig. 5.1 can be made more concise by initializing **`counter`** to **`0`** and replacing the **`while`** structure with

```
while ( ++counter <= 5 )   // repetition condition
   Console.WriteLine( counter );
```

This code saves a statement and eliminates the need for braces around the loop's bod,y because the incrementing occurs directly in the **`while`** condition before the condition is tested (remember that the precedence of **`++`** is higher than **`<=`**).

Good Programming Practice 5.1

Control counting loops with integer values.

Good Programming Practice 5.2

Place a blank line before and after each major control structure to make it stand out in the program.

Good Programming Practice 5.3

Vertical spacing above and below control structures, and indentation of the bodies of control structures within the control structure headers, gives programs a two-dimensional appearance that enhances readability.

5.3 `for` Repetition Structure

The **`for`** repetition structure handles the details of counter-controlled repetition. To illustrate the power of **`for`**, let us rewrite the program in Fig. 5.1. The result is displayed in Fig. 5.2.

The **`Main`** method (lines 8–14) operates as follows: When the **`for`** structure (line 12) begins executing, the program initializes the control variable **`counter`** to **`1`** (the first two elements of counter-controlled repetition—control variable *name* and *initial value*). Next, the program tests the loop-continuation condition, **`counter <= 5`**. The initial value of **`counter`** is **`1`**, thus the condition is true, so line 13 outputs the **`counter`**’s value. Then, the program increments variable **`counter`** in the expression **`counter++`**, and the loop begins again with the loop-continuation test. The control variable is now equal to **`2`**. This value does not exceed the final value, so the program performs the body statement again (i.e., performs the next iteration of the loop). This process continues until the control variable **`counter`** becomes **`6`**, causing the loop-continuation test to fail and repetition to terminate. The program continues by performing the first statement after the **`for`** structure. (In this case, method **`Main`** terminates because the program reaches the end of **`Main`**’s body.)

```
 1  // Fig. 5.2: ForCounter.cs
 2  // Counter-controlled repetition with the for structure.
 3
 4  using System;
 5
 6  class ForCounter
 7  {
 8     static void Main( string[] args )
 9     {
10        // initialization, repetition condition and incrementing
11        // are all included in the for structure
12        for ( int counter = 1; counter <= 5; counter++ )
13           Console.WriteLine( counter );
14     }
15  }
```

Fig. 5.2 Counter-controlled repetition with the **`for`** structure. (Part 1 of 2.)

```
1
2
3
4
5
```

Fig. 5.2 Counter-controlled repetition with the `for` structure. (Part 2 of 2.)

Figure 5.3 takes a closer look at the `for` structure in Fig. 5.2. The first line of the `for` structure (including the keyword `for` and everything in parentheses after `for`) sometimes is called the *`for` structure header*. Notice that the `for` structure specifies each of the items needed for counter-controlled repetition with a control variable. If there is more than one statement in the body of the `for`, braces (`{` and `}`) are required to define the loop's body.

Figure 5.2 uses the loop-continuation condition `counter <= 5`. If the programmer incorrectly writes `counter < 5`, the loop executes only four times. This common logic error is called an *off-by-one error*.

Common Programming Error 5.1

Using an incorrect relational operator or using an incorrect final value for a loop counter in the condition of a `while`, `for` or `do`/`while` structure (introduced in Section 5.6) can cause an off-by-one error.

Common Programming Error 5.2

Floating-point values may be approximate, so controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.

Testing and Debugging Tip 5.1

Using the final value in the condition of a `while` or `for` structure and using the `<=` relational operator will help avoid off-by-one errors. For a loop used to print the values from 1 to 10, for example, the loop-continuation condition should be `counter <= 10`, rather than `counter < 10` (which is an off-by-one error) or `counter < 11` (which also works). This approach is commonly known as one-based counting. *When we study arrays in Chapter 7, Arrays, we will see when programmers prefer* zero-based counting, *in which to count 10 times through a loop, `counter` is initialized to zero and the loop-continuation test is `counter < 10`.*

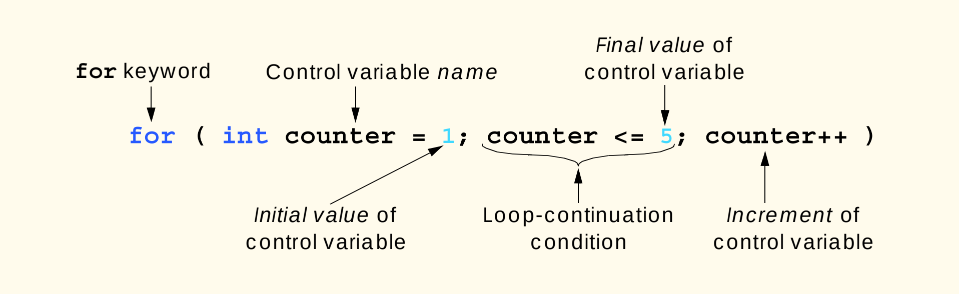

for keyword
Control variable name
Final value of control variable
for (int counter = 1; counter <= 5; counter++)
Initial value of control variable
Loop-continuation condition
Increment of control variable

Fig. 5.3 Components of a typical `for` header.

The general format of the `for` structure is

```
for ( expression1; expression2; expression3 )
   statement
```

where *expression1* names the loop's control variable and provides its initial value, *expression2* is the loop-continuation condition (containing the control variable's final value) and *expression3* increments or decrements the control variable. In most cases, the `for` structure can be represented with an equivalent `while` structure, with *expression1*, *expression2* and *expression3* placed as follows:

```
expression1;

while ( expression2 )
{
   statement
   expression3;
}
```

In Section 5.7, we discuss an exception to this rule.

In C#, programmers may declare the control variable in *expression1* of the `for` structure header (i.e., the control variable's type is specified before the variable name), rather than earlier in the code. When this occurs, the control variable can be used only in the body of the `for` structure (i.e., the name of the control variable will be unknown outside the `for` structure). Such a restriction on the use of a control variable name defines the variable's *scope*. The scope of a variable defines where it can be used in a program. Scope is discussed in detail in Chapter 6, Methods.

Common Programming Error 5.3

When a `for` structure declares its control variable in the initialization section of the `for` structure header, using the control variable after the `for` structure's body is a compiler error.

Sometimes, *expression1* and *expression3* in a `for` structure are comma-separated lists of expressions that enable the programmer to use multiple initialization expressions and/or multiple increment or decrement expressions. For example, there may be several control variables in a single `for` structure that must be initialized and incremented or decremented.

Good Programming Practice 5.4

Place only expressions involving control variables in the initialization and increment or decrement sections of a `for` structure. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the loop body (if they execute once per iteration of the loop, like incrementing or decrementing statements).

The three expressions in the `for` structure are optional. If *expression2* is omitted, C# assumes that the loop-continuation condition is always true, thus creating an infinite loop. A programmer might omit *expression1* if the program initializes the control variable before the loop. *Expression3* might be omitted if statements in the body of the `for` calculate the increment or decrement, or if no increment or decrement is necessary. The increment (or decrement) expression in the `for` structure acts as if it were a standalone statement at the end of the `for` body. Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

are equivalent when used in *expression3*. Some programmers prefer the form `counter++`, because the control variable increment occurs after the loop body executes. For this reason, the postincrementing (or postdecrementing) form in which the variable is incremented after it is used seems more natural. Because the variable being either incremented or decremented does not appear in a larger expression, preincrementing and postincrementing the variable have the same effect. The two semicolons in the `for` structure are required.

Common Programming Error 5.4

Using commas in a `for` *structure header instead of the two required semicolons is a syntax error.*

Common Programming Error 5.5

Placing a semicolon immediately to the right of a `for` *structure header's right parenthesis makes the body of that* `for` *structure an empty statement. This is normally a logic error.*

The initialization, loop-continuation condition and increment or decrement portions of a `for` structure can contain arithmetic expressions. For example, assume that `x = 2` and `y = 10`. If `x` and `y` are not modified in the loop body, the statement

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to the statement

```
for ( int j = 2; j <= 80; j += 5 )
```

The "increment" of a `for` structure may be negative, in which case it is really a decrement and the loop actually counts downward.

If the loop-continuation condition in the `for` structure is initially false, the body of the `for` structure does not execute. Instead, execution proceeds with the statement that follows the `for` structure.

The control variable frequently is printed or used in calculations in the body of a `for` structure, but it does not have to be. Often the control variable simply controls repetition and is not mentioned in the body of the `for` structure.

Testing and Debugging Tip 5.2

Avoid changing the value of the control variable in the body of a `for` *loo, to avoid subtle errors.*

The `for` structure flowchart is similar to that of the `while` structure. For example, the flowchart of the `for` structure in Fig. 5.2 appears in Fig. 5.4. This flowchart clarifies that the initialization occurs only once, and that incrementing occurs each time *after* the body statement is performed. Note that (besides small circles and flowlines) the flowchart contains only rectangle symbols and a diamond symbol. The rectangles and diamonds are filled with actions and decisions appropriate to the algorithm.

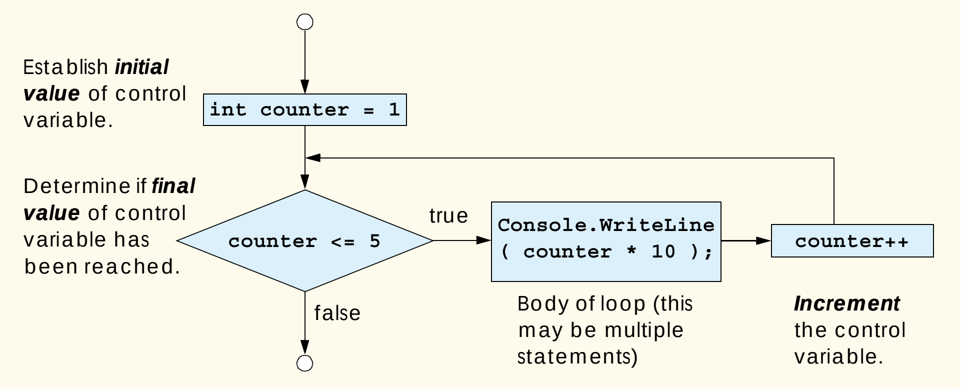

Establish *initial value* of control variable.
int counter = 1
Determine if *final value* of control variable has been reached.
counter <= 5
true
false
Console.WriteLine (counter * 10);
counter++
Body of loop (this may be multiple statements)
Increment the control variable.

Fig. 5.4 Flowcharting a typical `for` repetition structure.

5.4 Examples Using the `for` Structure

The following examples demonstrate methods of varying the control variable in a `for` structure. In each case, we write the appropriate `for` header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from `1` to `100` in increments of `1`.

```
for ( int i = 1; i <= 100; i++ )
```

b) Vary the control variable from `100` to `1` in increments of `-1` (decrements of `1`).

```
for ( int i = 100; i >= 1; i-- )
```

c) Vary the control variable from `7` to `77` in steps of `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from `20` to `2` in steps of `-2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the sequence of the following values: `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for ( int j = 2; j <= 20; j += 3 )
```

f) Vary the control variable over the sequence of the following values: `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for ( int j = 99; j >= 0; j -= 11 )
```

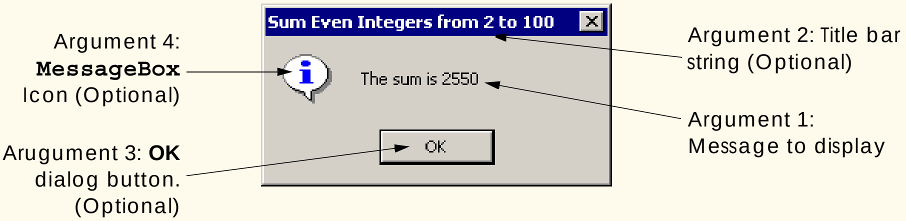
Common Programming Error 5.6

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` *in a loop counting down to 1) is usually a logic error that will yield incorrect results when the program runs.*

The next two sample programs demonstrate simple applications of the **`for`** repetition structure. The program in Fig. 5.5 uses the **`for`** structure to sum all the even integers from **`2`** to **`100`**, then displays the result in a **`MessageBox`**. Remember that to use **`MessageBox`**, you must add a reference to **`System.Windows.Forms.dll`** to your project, as explained in Chapter 3 (Section 3.2).

Figure 5.5 uses a version of method **`MessageBox.Show`** (lines 16–19) that takes four arguments. The dialog in the output of Fig. 5.5 illustrates the four arguments. As with the version that takes one argument, the first argument is the message to display. The second argument is the string to display in the dialog's title bar. The third argument is a value indicating which button(s) to display. The fourth argument indicates which icon to display to the left of the message. The MSDN documentation provided with Visual Studio .NET includes the complete listing of **`MessageBoxButtons`** and **`MessageBoxIcon`** choices. Figure 5.6 describes the message-dialog icons and Fig. 5.7 describes the message-dialog buttons.

```
 1  // Fig. 5.5: Sum.cs
 2  // Summation with the for structure.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  class Sum
 8  {
 9     static void Main( string[] args )
10     {
11        int sum = 0;
12
13        for ( int number = 2; number <= 100; number += 2 )
14           sum += number;
15
16        MessageBox.Show( "The sum is " + sum,
17           "Sum Even Integers from 2 to 100",
18           MessageBoxButtons.OK,
19           MessageBoxIcon.Information );
20
21     } // end method Main
22
23  } // end class Sum
```


Sum Even Integers from 2 to 100

The sum is 2550

OK
Argument 4: MessageBox Icon (Optional)
Arugument 3: OK dialog button. (Optional)
Argument 2: Title bar string (Optional)
Argument 1: Message to display

Fig. 5.5 Summation using **`for`**.

MessageBox Icons	Icon	Description
`MessageBoxIcon.Exclamation`	⚠	Specifies an exclamation point icon. Typically used to caution the user against potential problems.
`MessageBoxIcon.Information`	ℹ	Specifies that the dialog contains an informational message for the user.
`MessageBoxIcon.Question`	?	Specifies a question mark icon. Typically used in dialogs that ask the user a question.
`MessageBoxIcon.Error`	✖	Specifies a dialog with an **×** in a red circle. Alerts user of errors or important messages.

Fig. 5.6 Icons for message dialogs.

MessageBox Buttons	Description
`MessageBoxButton.OK`	Specifies that the dialog should include an **OK** button.
`MessageBoxButton.OKCancel`	Specifies that the dialog should include **OK** and **Cancel** buttons. Warns the user about some condition and allows the user to either continue or cancel an operation.
`MessageBoxButton.YesNo`	Specifies that the dialog should contain **Yes** and **No** buttons. Used to ask the user a question.
`MessageBoxButton.YesNoCancel`	Specifies that the dialog should contain **Yes**, **No** and **Cancel** buttons. Typically used to ask the user a question but still allows the user to cancel the operation.
`MessageBoxButton.RetryCancel`	Specifies that the dialog should contain **Retry** and **Cancel** buttons. Typically used to inform a user about a failed operation and allow the user to retry or cancel the operation.
`MessageBoxButton.AbortRetryIgnore`	Specifies that the dialog should contain **Abort**, **Retry** and **Ignore** buttons. Typically used to inform the user that one of a series of operations has failed and allow the user to abort the series of operations, retry the failed operation or ignore the failed operation and continue.

Fig. 5.7 Buttons for message dialogs.

The body of the `for` structure in Fig. 5.5 actually could be merged into the rightmost portion of the `for` header by using a *comma* as follows:

```
for ( int number = 2; number <= 100;
      sum += number, number += 2)
   ; // empty statement
```

Similarly, the initialization `sum = 0` could be merged into the initialization section of the `for` structure. Statements that precede a `for` and statements in the body of a `for` often can be merged into the `for` header. However, such merging could decrease the readability of the program.

Good Programming Practice 5.5

Limit the size of control structure headers to a single line if possible.

The next example uses a `for` structure to compute compound interest. Consider the following problem statement:

> *A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. To determine these amounts, use the following formula:*
>
> $a = p\,(1 + r)^{n}$
>
> *where*
>
> p is the original amount invested (i.e., the principal)
> r is the annual interest rate
> n is the number of years
> a is the amount on deposit at the end of the nth year.

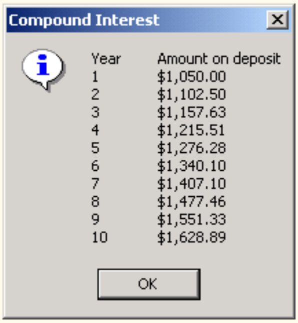
This problem involves a loop that performs the indicated calculation for each of the 10 years that the money remains on deposit. A solution is the program shown in Fig. 5.8.

Line 11 in method `Main` declares two ***`decimal`*** variables—`amount` and `principal`—and initializes `principal` to `1000.00`. The type `decimal` is a primitive data type used for monetary calculations. C# treats such constants as the `1000.00` in Fig. 5.8 as type `double`. Similarly, C# treats whole number constants, like 7 and -22, as having type `int`. Values of type `double` cannot be converted implicitly to type `decimal`, so we use a cast operator to convert the `double` value 1000.00 to type `decimal`. It also is possible to specify that a constant is of type `decimal` by appending the letter `m` to the constant, as in 1000.0m. Line 12 declares `double` variable `rate`, which we initialize to `.05`.

```
1  // Fig. 5.8: Interest.cs
2  // Calculating compound interest.
3
4  using System;
5  using System.Windows.Forms;
6
7  class Interest
8  {
```

Fig. 5.8 Calculating compound interest with `for`. (Part 1 of 2.)

```
 9     static void Main( string[] args )
10     {
11        decimal amount, principal = ( decimal ) 1000.00;
12        double rate = .05;
13        string output;
14
15        output = "Year\tAmount on deposit\n";
16
17        for ( int year = 1; year <= 10; year++ )
18        {
19           amount = principal *
20              ( decimal ) Math.Pow( 1.0 + rate, year );
21
22           output += year + "\t" +
23              String.Format( "{0:C}", amount ) + "\n";
24        }
25
26        MessageBox.Show( output, "Compound Interest",
27           MessageBoxButtons.OK, MessageBoxIcon.Information );
28
29     } // end method Main
30
31  } // end class Interest
```



```
Year	Amount on deposit
1	$1,050.00
2	$1,102.50
3	$1,157.63
4	$1,215.51
5	$1,276.28
6	$1,340.10
7	$1,407.10
8	$1,477.46
9	$1,551.33
10	$1,628.89
```

Fig. 5.8 Calculating compound interest with **`for`**. (Part 2 of 2.)

The **`for`** structure executes its body 10 times, varying control variable **`year`** from **`1`** to **`10`** in increments of **`1`**. Note that **`year`** represents *n* in the problem statement. C# does not have an exponentiation operator, so we use **`static`** method **`Pow`** in class **`Math`** for this purpose. **`Math.Pow( x, y )`** calculates the value of **`x`** raised to the **`y`**th power. Method **`Math.Pow`** takes two arguments of type **`double`** and returns a **`double`** value. Lines 19–20 perform the calculation from the problem statement

$$a = p\,(1 + r)^{n}$$

where *a* is **`amount`**, *p* is **`principal`**, *r* is **`rate`** and *n* is **`year`**.

Lines 22–23 append additional text to the end of the string **`output`**. The text includes the current **`year`** value, a tab character to position to the second column, the result of the method call `String.Format( "{0:C}", amount )` and a newline character to position to the next line. The call to method `String.Format` converts `amount` to a `string` and formats this `string` so that it will display with two decimal places. [*Note*: Method `Format` uses the string formatting codes to represent numeric and monetary values in a form that is appropriate to the execution environment. For example, in the US, monetary values are formatted with two digits to the right of the decimal point and the thousands separator is a comma.] The first argument is the format string. We have already seen such strings in the form `{0}`, `{1}` and so on. In those cases, the digit indicated the argument being displayed. In more complicated format strings, such as the one shown in this example (`"{0:C}"`), the first digit (`0`) still represents the argument to display. The information specified after the colon (`:`) specifies the formatting of the argument, and usually is called the *formatting code*. In this case, we are using formatting code `C`, which indicates that our string should be displayed as a monetary amount with two digits after the decimal point. There are several other formatting codes, which can be found in the MSDN documentation. Figure 5.9 shows several formatting codes.

The variables `amount` and `principal` were declared to be of type `decimal` because the program deals with fractional parts of dollars. In such cases, programs need a type that allows decimal points in its values. Variable `rate` is of type `double` because it is used in the calculation `1.0 + rate`, which appears as a `double` argument to the `Pow` method of class `Math`. Note that the calculation `1.0 + rate` appears in the body of the `for` statement. The calculation produces the same result each time through the loop, so repeating the calculation is unnecessary.

Performance Tip 5.1

Avoid placing expressions with values that do not change inside a loop. Such expressions should be evaluated once before the loop. Most good compilers will fix this automatically with a process that compilers perform called optimization.

Format Code	Description
`C` or `c`	Formats the string as currency. Precedes the number with an appropriate currency symbol (`$` in the US). Separates digits with an appropriate separator character (comma in the US) and sets the number of decimal places to two by default.
`D` or `d`	Formats the string as a decimal. Displays number as an integer.
`N` or `n`	Formats the string with commas and two decimal places.
`E` or `e`	Formats the number using scientific notation with a default of six decimal places.
`F` or `f`	Formats the string with a fixed number of decimal places (two by default).
`G` or `g`	General. Either `E` or `F`.
`X` or `x`	Formats the string as hexadecimal.

Fig. 5.9 `string` formatting codes.

5.5 `switch` Multiple-Selection Structure

The previous chapter discussed the `if` single-selection and the `if`/`else` double-selection structures. Occasionally, an algorithm contains a series of decisions in which the algorithm tests a variable or expression separately for each *constant integral expression* or *constant string expression* the variable or expression may assume. A constant integral expression is any expression involving character and integer constants that evaluates to an integer value (i.e., values of type `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` and `char`). A constant string expression is any expression composed of string literals that always results in the same `string`. The algorithm then takes different actions based on those values. C# provides the *`switch` multiple-selection structure* to handle such decision making.

In the next example (Fig. 5.10), let us assume that a class of 10 students took an exam and that each student received a letter grade of A, B, C, D or F. The program will input the letter grades and summarize the results by using `switch` to count the number of each different letter grade that students earned on an exam. Line 10 declares variable `grade` as type `char`. Lines 11–15 define counter variables that the program uses to count each letter grade. This variable stores the user's input for each grade. Line 17 begins a `for` structure that loops 10 times. At each iteration, line 19 prompts the user for the next grade, and line 20 invokes `Char` method `Parse` to read the user input as a `char`. Nested in the body of the `for` structure is a `switch` structure (lines 22–56) that processes the letter grades. The `switch` structure consists of a series of *`case` labels* and an optional *`default` case*.

When the flow of control reaches the `switch` structure, the program evaluates the *controlling expression* (`grade` in this example) in the parentheses following keyword `switch`. The value of this expression is compared with each `case` label until a match occurs. Assume the user entered the letter `B` as the grade. `B` is compared to each `case` in the `switch`, until a match occurs at line 29 (`case 'B':`). When this happens, the statements for that `case` execute. For the letter `B`, lines 31–32 increment the number of `B` grades stored in variable `bCount`, and the `switch` structure exits immediately with the *`break` statement*. The `break` statement causes program control to proceed with the first statement after the `switch` structure. In this case, we reach the end of the `for` structure's body, so control flows to the control-variable increment expression in the `for` structure header. Then the counter variable in the `for` structure is incremented, and the loop-continuation condition is evaluated to determine whether another iteration of the loop is necessary.

```
1   // Fig. 5.10: SwitchTest.cs
2   // Counting letter grades.
3
4   using System;
5
6   class SwitchTest
7   {
8      static void Main( string[] args )
9      {
10        char grade;       // one grade
11        int aCount = 0,   // number of As
12            bCount = 0,   // number of Bs
13            cCount = 0,   // number of Cs
```

Fig. 5.10 Example using `switch`. (Part 1 of 3.)

```
          dCount = 0,  // number of Ds
          fCount = 0;  // number of Fs

      for ( int i = 1; i <= 10; i++ )
      {
         Console.Write( "Enter a letter grade: " );
         grade = Char.Parse( Console.ReadLine() );

         switch ( grade )
         {
            case 'A':   // grade is uppercase A
            case 'a':   // or lowercase a
               ++aCount;
               break;

            case 'B':   // grade is uppercase B
            case 'b':   // or lowercase b
               ++bCount;
               break;

            case 'C':   // grade is uppercase C
            case 'c':   // or lowercase c
               ++cCount;
               break;

            case 'D':   // grade is uppercase D
            case 'd':   // or lowercase d
               ++dCount;
               break;

            case 'F':   // grade is uppercase F
            case 'f':   // or lowercase f
               ++fCount;
               break;

            default:    // processes all other characters
               Console.WriteLine(
                  "Incorrect letter grade entered." +
                  "\nEnter a new grade" );
               break;

         } // end switch

      } // end for

      Console.WriteLine(
         "\nTotals for each letter grade are:\nA: {0}" +
         "\nB: {1}\nC: {2}\nD: {3}\nF: {4}", aCount, bCount,
         cCount, dCount, fCount );

   } // end method Main

} // end class SwitchTest
```

Fig. 5.10 Example using **`switch`**. (Part 2 of 3.)

```
Enter a letter grade: a
Enter a letter grade: A
Enter a letter grade: c
Enter a letter grade: F
Enter a letter grade: z
Incorrect letter grade entered.
Enter a new grade
Enter a letter grade: D
Enter a letter grade: d
Enter a letter grade: B
Enter a letter grade: a
Enter a letter grade: C
Enter a letter grade: C

Totals for each letter grade are:
A: 3
B: 1
C: 3
D: 2
F: 1
```

Fig. 5.10 Example using `switch`. (Part 3 of 3.)

Good Programming Practice 5.6

Indent the body statements of each `case` *in a* `switch` *structure.*

If no match occurs between the controlling expression’s value and a `case` label, the `default` case (line 49) executes. Lines 50–52 display an error message. Note that the `default` case is optional in the `switch` structure. If the controlling expression does not match a `case` and there is no `default` case, program control proceeds to the next statement after the `switch` structure. It is also important to understand that, in C#, only the statements for one `case` can be executed in one `switch` statement.

Each `case` can contain multiple actions or no actions at all. A `case` with no statements is considered an *empty* `case`, and can omit the `break` statement. The `break` statement is required for each `case` (including the `default` case) that contains statements. The last `case` in a `switch` structure must not be an empty `case`. If the `case` label for an empty `case` matches our controlling expression, *fall through* occurs. This means that the `switch` structure executes the statements in the next case. If that `case` is also empty, this process will continue until a nonempty case is found, and then that `case`’s statements will execute. This provides the programmer with a way to specify statements to executed for several labels. Figure 5.10 demonstrates this. Lines 26–27 execute for both cases on lines 24–25 (if the grade entered was either `A` or `a`), lines 31–32 execute for both cases on lines 29–30 (if the grade entered was either `B` or `b`) and so on.

Common Programming Error 5.7

Not including a `break` *statement at the end of each* `case` *in a* `switch` *is a syntax error. The exception to this rule is the empty* `case`*.*

Common Programming Error 5.8

Be sure to check all possible values when creating **`case`***s to confirm that no two* **`case`***s in a* **`switch`** *statement are for the same integral value. If the values are the same, a compile-time error will occur.*

Finally, it is important to notice that the **`switch`** structure is different from other structures in that braces are not required around multiple actions in a **`case`** of a **`switch`**. The general **`switch`** structure (using a **`break`** in each **`case`**) is flowcharted in Fig. 5.11.

Again, note that (besides small circles and flowlines) the flowchart contains only rectangle and diamond symbols. The programmer fills the rectangles and diamonds with actions and decisions appropriate to the algorithm. Although nested control structures are common, it is rare to find nested **`switch`** structures in a program.

Good Programming Practice 5.7

Provide a **`default`** *case in every* **`switch`** *structure. Cases not explicitly tested in a* **`switch`** *that lacks a* **`default`** *case are ignored. Including a* **`default`** *case focuses the programmer on processing exceptional conditions. There are situations, however, in which no* **`default`** *processing is required.*

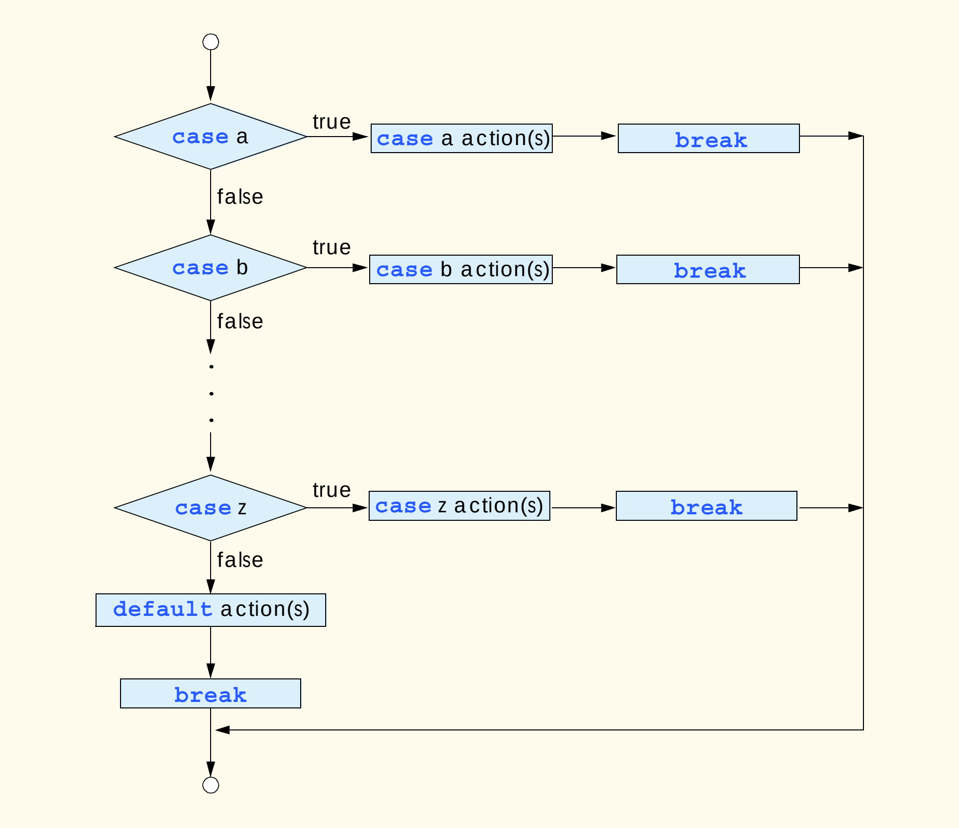


Fig. 5.11 Flowcharting the **`switch`** multiple-selection structure.

Good Programming Practice 5.8

Although the `case`*s in a* `switch` *structure can occur in any order, it is considered a good programming practice to place the* `default` *case last.*

When using the `switch` structure, remember that all cases in a particular switch structure must be either integral values or strings. A *character constant* is represented as a specific character in single quotes (such as `'A'`). An integer constant is simply an integer value. The expression after each `case` can also be a *constant variable*—a variable that contains a value that does not change throughout the entire program. Such a variable is declared with keyword ***`const`*** (discussed in Chapter 6, Methods).

Chapter 10, Object-Oriented Programming: Polymorphism, presents a more elegant way of implementing `switch` logic. We use a technique called polymorphism to create programs that are clearer and easier to maintain and extend than programs that use `switch` logic.

5.6 `do`/`while` Repetition Structure

The ***`do`/`while`*** repetition structure is similar to the `while` structure. In the `while` structure, the test of the loop-continuation condition occurs at the beginning of the loop, before the body of the loop executes. The `do`/`while` structure tests the loop-continuation condition *after* the loop body executes; therefore, *the loop body always executes at least once*. When a `do`/`while` structure terminates, execution continues with the statement after the `while` clause. The program in Fig. 5.12 uses a `do`/`while` structure to output the values 1–5.

```
// Fig. 5.12: DoWhileLoop.cs
// The do/while repetition structure.

using System;

class DoWhileLoop
{
   static void Main( string[] args )
   {
      int counter = 1;

      do
      {
         Console.WriteLine( counter );
         counter++;
      } while ( counter <= 5 );

   } // end method Main

} // end class DoWhileLoop
```

```
1
2
3
4
5
```

Fig. 5.12 Using the `do`/`while` repetition structure.

Lines 12–16 demonstrate the `do`/`while` structure. When program execution reaches the `do`/`while` structure, the program executes lines 14–15, which display the value of `counter` (at this point, `1`) and increment `counter` by `1`. Then, the program evaluates the condition on line 16. At this point, variable `counter` is 2, which is less than or equal to 5, so the `do`/`while` structure's body executes again. The fifth time the structure executes, line 14 outputs the value `5` and line 15 increments `counter` to `6`. Then the condition on line 16 evaluates to false and the `do`/`while` structure exits.

The `do`/`while` flowchart (Fig. 5.13) makes it clear that the loop-continuation condition does not execute until the body executes at least once. The flowchart contains only a rectangle and a diamond. The programmer fills the rectangle and diamond with actions and decisions appropriate to the algorithm.

Note that it is not necessary to use braces in the `do`/`while` structure if there is only one statement in the body. However, the braces normally are included to avoid confusion between the `while` and `do`/`while` structures. For example,

```
while ( condition )
```

typically is the header to a `while` structure. A `do`/`while` with no braces around the single statement body appears as

```
do
   statement
while ( condition );
```

which can be confusing. The last line—`while(` *condition* `);`—might be misinterpreted by the reader as a `while` structure containing an empty statement (the semicolon by itself). Thus, the `do`/`while` with one statement often is written as follows to avoid confusion:

```
do
{
   statement
} while ( condition );
```

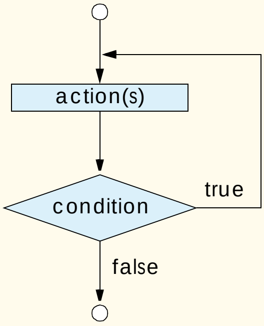


Fig. 5.13 Flowcharting the `do`/`while` repetition structure.

Good Programming Practice 5.9

Some programmers always include braces in a **`do`***/***`while`** *structure, even when the braces are unnecessary. This helps eliminate ambiguity between a* **`while`** *structure and a* **`do`***/***`while`** *structure that contains only one statement.*

Common Programming Error 5.9

Infinite loops occur when the loop-continuation condition in a **`while`***,* **`for`** *or* **`do`***/***`while`** *structure never becomes false. To prevent this, make sure there is no semicolon immediately after the header of a* **`while`** *or* **`for`** *structure or after the word* **`do`** *in a* **`do`***/***`while`** *statement. In a counter-controlled loop, make sure the control variable is incremented (or decremented) in the body of the loop. In a sentinel-controlled loop, make sure the sentinel value eventually is input.*

5.7 Statements `break` and `continue`

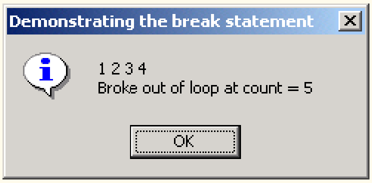
The ***`break`*** and ***`continue`*** statements alter the flow of control. The **`break`** statement, when executed in a **`while`**, **`for`**, **`do`**/**`while`** or **`switch`** structure, causes immediate exit from that structure. Execution continues with the first statement that follows the structure. Common uses of the **`break`** statement are to exit prematurely from a loop or to exit a **`switch`** structure (as in Fig. 5.10). Figure 5.14 demonstrates the **`break`** statement in a **`for`** repetition structure.

When the **`if`** structure in line 16 detects that **`count`** is **`5`**, **`break`** is executed. This terminates the **`for`** structure and the program proceeds to line 24 (immediately after the **`for`**). The string-concatenation statement produces the string that is displayed in the message dialog in lines 26–27. The loop executes its body only four times.

```
 1  // Fig. 5.14: BreakTest.cs
 2  // Using the break statement in a for structure.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  class BreakTest
 8  {
 9     static void Main( string[] args )
10     {
11        string output = "";
12        int count;
13
14        for ( count = 1; count <= 10; count++ )
15        {
16           if ( count == 5 )
17              break;           // skip remaining code in loop
18                               // if count == 5
19
20           output += count + " ";
21
22        } // end for loop
23
```

Fig. 5.14 Using the **`break`** statement in a **`for`** structure. (Part 1 of 2.)

```
      output += "\nBroke out of loop at count = " + count;

      MessageBox.Show( output, "Demonstrating the break statement",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method Main

} // end class BreakTest
```



```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 5.14 Using the `break` statement in a `for` structure. (Part 2 of 2.)

The `continue` statement, when executed in a `while`, `for` or `do`/`while` structure, skips the remaining statements in the body of that structure and proceeds with the next iteration of the loop. In `while` and `do`/`while` structures, the loop-continuation condition evaluates immediately after `continue` executes. In a `for` structure, the increment/decrement expression executes, then the loop-continuation test evaluates.

We have stated that the `while` structure can replace the `for` structure in most cases. One exception occurs when the increment/decrement expression in the `while` structure follows the `continue` statement. In this case, the increment/decrement does not execute before the repetition-continuation condition is tested, and the `while` does not execute in the same manner as the `for`.

Figure 5.15 uses the `continue` statement in a `for` structure to skip the string-concatenation statement on line 19 when the `if` structure (line 15) determines that the value of `count` is `5`. When the `continue` statement executes, program control continues with the increment of the control variable in the `for` structure.

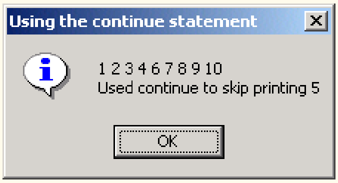
Good Programming Practice 5.10

Some programmers believe that ***`break`*** *and* ***`continue`*** *violate structured programming. The effects of these statements can be achieved by structured programming techniques, so these programmers avoid* ***`break`*** *and* ***`continue`****.*

```
1  // Fig. 5.15: ContinueTest.cs
2  // Using the continue statement in a for structure.
3
4  using System;
5  using System.Windows.Forms;
6
7  class ContinueTest
8  {
9     static void Main( string[] args )
10    {
11       string output = "";
```

Fig. 5.15 Using the `continue` statement in a `for` structure.

```

      for ( int count = 1; count <= 10; count++ )
      {
         if ( count == 5 )
            continue;       // skip remaining code in loop
                            // only if count == 5

         output += count + " ";
      }

      output += "\nUsed continue to skip printing 5";

      MessageBox.Show( output, "Using the continue statement",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method Main

} // end class ContinueTest
```



```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.15 Using the **`continue`** statement in a **`for`** structure.

Performance Tip 5.2

When used properly, the **`break`** *and* **`continue`** *statements perform faster than their corresponding structured techniques.*

Software Engineering Observation 5.1

There is a debate between achieving quality software engineering and achieving the best performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following "rule of thumb": First, make your code simple and correct; then make it fast and small, but only if necessary.

5.8 Logical and Conditional Operators

So far, we have studied only *simple conditions*, such as **`count <= 10`**, **`total > 1000`** and **`number != sentinelValue`**. These conditions were expressed in terms of the relational operators **`>`**, **`<`**, **`>=`** and **`<=`** and the equality operators **`==`** and **`!=`**. Each decision tested one condition. To test multiple conditions in the process of making a decision, we performed these tests in separate statements or in nested **`if`** or **`if`**/**`else`** structures.

C# provides several *logical and conditional operators* that may be used to form complex conditions by combining simple conditions. The operators are **`&&`** (*conditional AND*), **`&`** (*logical AND*), **`||`** (*conditional OR*), **`|`** (*logical OR*), **`^`** (*logical exclusive OR or logical XOR*) and **`!`** (*logical NOT*, also called *logical negation*). We will consider examples using each of these operators.

Common Programming Error 5.10

Placing a space between the `&&` or `||` operator results in a syntax error.

Suppose we wish to ensure that two conditions are *both* true in a program before we choose a certain path of execution. In this case, we can use the conditional `&&` operator as follows:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

This `if` statement contains two simple conditions. The condition `gender == 1` might be evaluated to determine whether a person is female. The condition `age >= 65` is evaluated to determine whether a person is a senior citizen. The two simple conditions are evaluated first, because the precedences of `==` and `>=` are both higher than the precedence of `&&`. The `if` statement then considers the combined condition

```
gender == 1 && age >= 65
```

This condition is true *if and only if* both the simple conditions are true. Finally, if this combined condition is true, the body statement increments the count of `seniorFemales` by `1`. If either or both of the simple conditions are false, the program skips the incrementing and proceeds to the statement that follows the `if` structure. The preceding combined condition can be made more readable by adding redundant parentheses:

```
( gender == 1 ) && ( age >= 65 )
```

The table in Fig. 5.16 summarizes the `&&` operator. The table shows all four possible combinations of false and true values for *expression1* and *expression2*. Such tables often are called *truth tables*. C# evaluates to true or false expressions that include relational operators, equality operators, logical operators and/or conditional operators.

Now let us consider the `||` (conditional OR) operator. Suppose we wish to ensure that either *or* both of two conditions are true before we choose a certain path of execution. We use the `||` operator in the following program segment:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   Console.WriteLine( "Student grade is A" );
```

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 5.16 Truth table for the `&&` (conditional AND) operator.

which also contains two simple conditions. The condition `semesterAverage >= 90` determines whether the student deserves an “A” in the course because of a solid performance throughout the semester. The condition `finalExam >= 90` determines whether the student deserves an “A” in the course because of an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
semesterAverage >= 90 || finalExam >= 90
```

and awards the student an “A” if either or both of the simple conditions are true. Note that the message “`Student grade is A`” prints unless *both* of the simple conditions are false. Figure 5.17 is a truth table for the conditional OR operator (`||`).

The `&&` operator has a higher precedence than the `||` operator. Both operators associate from left to right. An expression containing `&&` or `||` operators is evaluated only until truth or falsity is known. Thus, evaluation of the expression

```
gender == 1 && age >= 65
```

stops immediately if `gender` is not equal to `1` (i.e., the entire expression is false) and continue if `gender` is equal to `1` (i.e., the entire expression is true, even if the condition `age >= 65` is true). This performance feature for the evaluation of conditional AND and conditional OR expressions is called *short-circuit evaluation.*

Performance Tip 5.3

In expressions using operator `&&`*, if the separate conditions are independent of one another, make the condition most likely to be false the leftmost condition. In expressions using operator* `||`*, make the condition most likely to be true the leftmost condition. This use of short-circuit evaluation can reduce a program’s execution time.*

The *logical AND* (`&`) and *logical OR* (`|`) operators are similar to the conditional AND and conditional OR operators, with one exception—the logical operators always evaluate both of their operands (i.e., there is no short-circuit evaluation). Therefore, the expression

```
gender == 1 & age >= 65
```

evaluates `age >= 65`, regardless of whether `gender` is equal to `1`. This is useful if the right operand of the logical AND or logical OR operator includes a needed *side effect*—a modification of a variable’s value. For example, the expression

```
birthday == true | ++age >= 65
```

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 5.17 Truth table for the `||` (conditional OR) operator.

guarantees that the condition `++age >= 65` evaluates and increments the variable `age` in the preceding expression, regardless of whether the overall expression is true or false. Likewise, if we want the condition in the right operand to be the result of a method call, and we want the method to execute in any case, then we can use the `|` operator.

Common Programming Error 5.11

Avoid expressions with side effects in conditions. The side effects might look clever, but they often cause subtle errors and can be confusing to other people reading or maintaining your code.

A condition containing the *logical exclusive OR* (`^`) operator is true *if and only if one of its operands results in a true value and one results in a false value*. If both operands are true or both are false, the result of the entire condition is false. Figure 5.18 is a truth table for the logical exclusive OR operator (`^`). This operator evaluates both of its operands (i.e., there is no short-circuit evaluation).

C# provides the `!` (logical negation) operator to enable a programmer to “reverse” the meaning of a condition. Unlike the logical operators `&&`, `&`, `||`, `|` and `^`, which combine two conditions (binary operators), the logical negation operator has only a single condition as an operand (unary operator). The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is false. This is demonstrated by the following program segment:

```
if ( ! ( grade == sentinelValue ) )
   Console.WriteLine( "The next grade is " + grade );
```

The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator. Figure 5.19 is a truth table for the logical negation operator.

expression1	**expression2**	**expression1 ^ expression2**
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

Fig. 5.18 Truth table for the logical exclusive OR (`^`) operator.

expression	**!expression**
`false`	`true`
`true`	`false`

Fig. 5.19 Truth table for operator `!` (logical NOT).

In most cases, the programmer can avoid using logical negation by expressing the condition differently with relational or equality operators. For example, the preceding statement may also be written as follows:

```
if ( grade != sentinelValue )
   Console.WriteLine( "The next grade is " + grade );
```

This flexibility can help a programmer express a condition more naturally.

The console application in Fig. 5.20 demonstrates all the conditional and logical operators by displaying their truth tables in a label.

```
// Fig. 5.20: LogicalOperators.cs
// Demonstrating the logical operators.
using System;

class LogicalOperators
{
   // main entry point for application
   static void Main( string[] args )
   {
      // testing the conditional AND operator (&&)
      Console.WriteLine( "Conditional AND (&&)" +
         "\nfalse && false: " + ( false && false ) +
         "\nfalse && true:  " + ( false && true ) +
         "\ntrue && false:  " + ( true && false ) +
         "\ntrue && true:   " + ( true && true ) );

      // testing the conditional OR operator (||)
      Console.WriteLine( "\n\nConditional OR (||)" +
         "\nfalse || false: " + ( false || false ) +
         "\nfalse || true:  " + ( false || true ) +
         "\ntrue || false:  " + ( true || false ) +
         "\ntrue || true:   " + ( true || true ) );

      // testing the logical AND operator (&)
      Console.WriteLine( "\n\nLogical AND (&)" +
         "\nfalse & false: " + ( false & false ) +
         "\nfalse & true:  " + ( false & true ) +
         "\ntrue & false:  " + ( true & false ) +
         "\ntrue & true:   " + ( true & true ) );

      // testing the logical OR operator (|)
      Console.WriteLine( "\n\nLogical OR (|)" +
         "\nfalse | false: " + ( false | false ) +
         "\nfalse | true:  " + ( false | true ) +
         "\ntrue | false:  " + ( true | false ) +
         "\ntrue | true:   " + ( true | true ) );

      // testing the logical exclusive OR operator (^)
      Console.WriteLine( "\n\nLogical exclusive OR (^)" +
         "\nfalse ^ false: " + ( false ^ false ) +
         "\nfalse ^ true:  " + ( false ^ true ) +
```

Fig. 5.20 Demonstrating the conditional and logical operators. (Part 1 of 2.)

```
42           "\ntrue ^ false:  " + ( true ^ false ) +
43           "\ntrue ^ true:   " + ( true ^ true ) );
44
45        // testing the logical NOT operator (!)
46        Console.WriteLine( "\n\nLogical NOT (!)" +
47           "\n!false: " + ( !false ) +
48           "\n!true:  " + ( !true ) );
49     }
50  }
```

```
Conditional AND (&&)
false && false: False
false && true:  False
true && false:  False
true && true:   True

Conditional OR (||)
false || false: False
false || true:  True
true || false:  True
true || true:   True

Logical AND (&)
false & false: False
false & true:  False
true & false:  False
true & true:   True

Logical OR (|)
false | false: False
false | true:  True
true | false:  True
true | true:   True

Logical exclusive OR (^)
false ^ false: False
false ^ true:  True
true ^ false:  True
true ^ true:   False

Logical NOT (!)
!false: True
!true:  False
```

Fig. 5.20 Demonstrating the conditional and logical operators. (Part 2 of 2.)

On line 11, we begin class **`LogicalOperators`**. Method **`Main`** (lines 8–49) contains the code for this program. Lines 11–15 demonstrate the **`&&`** operator; lines 25–29 demonstrate the **`&`** operator. Notice that, to display one **`&`** symbol, two **`&`**s are required. The remainder of the constructor demonstrates the **`||`**, **`|`**, **`^`** and **`!`** operators.

When a `bool` value is concatenated to a `string`, C# adds the string representation of the boolean value, which will be either `"False"` or `"True"`.

Figure 5.21 shows the precedence and associativity of the C# operators introduced to this point. The operators are shown from top to bottom in decreasing order of precedence.

5.9 Structured-Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture is, and our collective wisdom is considerably sparser. We have learned that structured programming produces programs that are easier to understand, test, debug, modify and prove correct in a mathematical sense than unstructured programs.

Figure 5.22 summarizes C#'s control structures. Small circles in the figure indicate the single entry point and the single exit point of each structure. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form only a limited set of control structures and to build structured programs by combining control structures in only two simple ways.

For simplicity, only single-entry/single-exit control structures are used—there is only one way to enter and only one way to exit each control structure. To connect control structures in sequence to form structured programs, the exit point of one control structure is connected to the entry point of the next control structure (i.e., the control structures are simply placed one after another in a program). We call this process "control-structure stacking." The rules for forming structured programs also allow control structures to be nested. Figure 5.23 contains the rules for forming properly structured programs. The rules assume that the rectangle flowchart symbol can indicate any action, including input/output.

Operators	**Associativity**	**Type**
`()`	left to right	parentheses
`++ --`	right to left	unary postfix
`++ -- + - !` (*type*)	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&`	left to right	logical AND
`^`	left to right	logical exclusive OR
`|`	left to right	logical inclusive OR
`&&`	left to right	conditional AND
`||`	left to right	conditional OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 5.21 Precedence and associativity of the operators discussed so far.

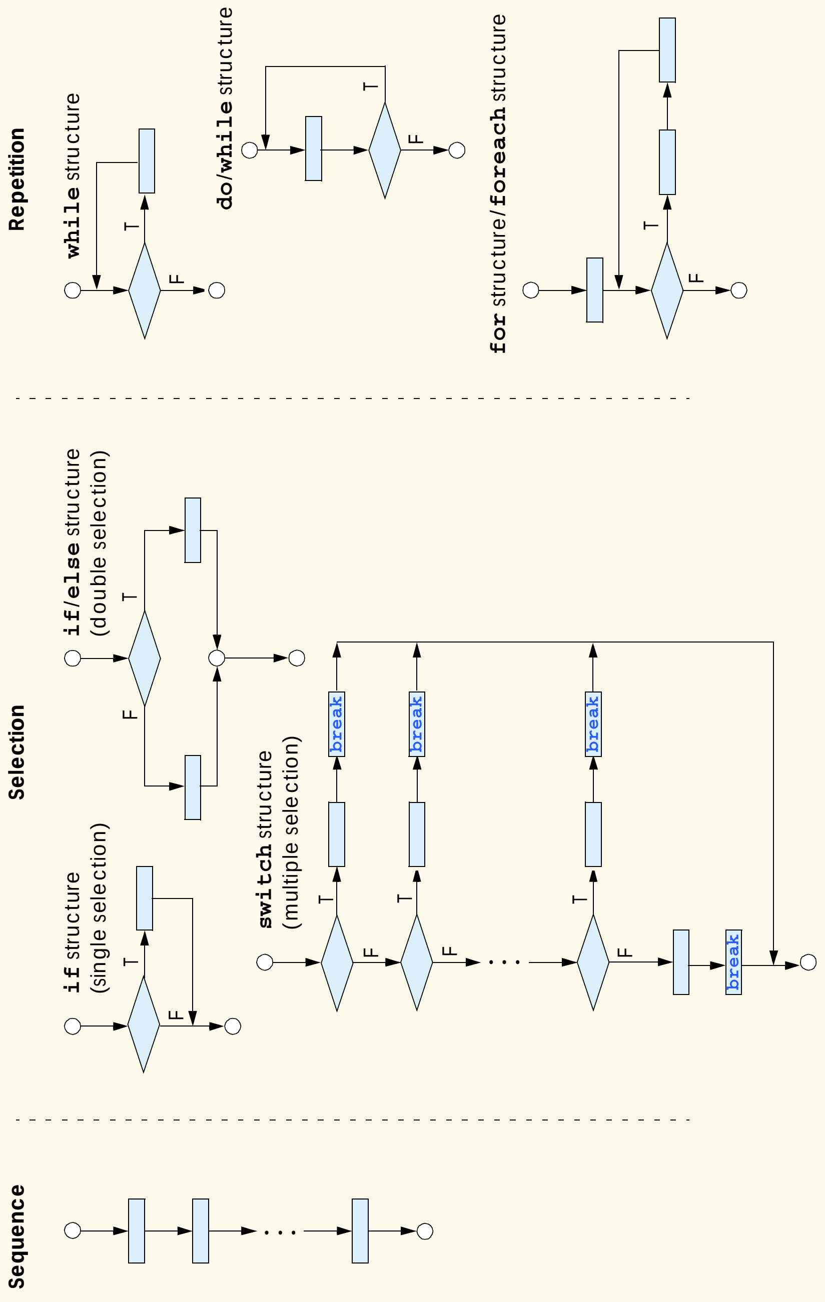

Sequence
Selection
Repetition
if structure (single selection)
if/else structure (double selection)
switch structure (multiple selection)
while structure
do/while structure
for structure/foreach structure
T
F
break
break
break
break
. . .

Fig. 5.22 C#'s single-entry/single-exit sequence, selection and repetition structures.

Rules for Forming Structured Programs

1) Begin with the "simplest flowchart" (Fig. 5.24).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3) Any rectangle (action) can be replaced by any control structure (sequence, `if`, `if`/`else`, `switch`, `while`, `do`/`while`, `for` or `foreach`, as we will see in Chapter 7, Arrays).
4) Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 5.23 Rules for forming structured programs.

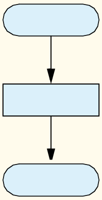


Fig. 5.24 Simplest flowchart.

Applying the rules of Fig. 5.23 always results in a structured flowchart with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest flowchart results in a structured flowchart that contains many rectangles in sequence (Fig. 5.25). Notice that rule 2 generates a stack of control structures; therefore, we call rule 2 the *stacking rule*.

Rule 3 is the *nesting rule*. Repeatedly applying rule 3 to the simplest flowchart results in a flowchart with neatly nested control structures. For example, in Fig. 5.26, the rectangle in the simplest flowchart first is replaced with a double-selection (`if`/`else`) structure. Then rule 3 is applied again to both rectangles in the double-selection structure, replacing each of the rectangles with a double-selection structure. The dashed boxes around each of the double-selection structures represent the rectangles that were replaced with these structures.

Good Programming Practice 5.11

Too many levels of nesting can make a program difficult to understand. As a general rule, try to avoid using more than three levels of nesting.

Rule 4 generates larger, more involved and deeply-nested structures. The flowcharts that emerge from applying the rules in Fig. 5.23 constitute the set of all possible structured flowcharts and the set of all possible structured programs.The structured approach has the advantage of using only eight simple single-entry/single-exit pieces and allowing us to assemble them in only two simple ways. Figure 5.27 shows the kinds of correctly stacked building blocks that emerge from applying rule 2 and the kinds of correctly nested building blocks that emerge from applying rule 3. The figure also shows the kind of overlapped building blocks that cannot occur in structured flowcharts (as a result of avoiding `goto` statements).

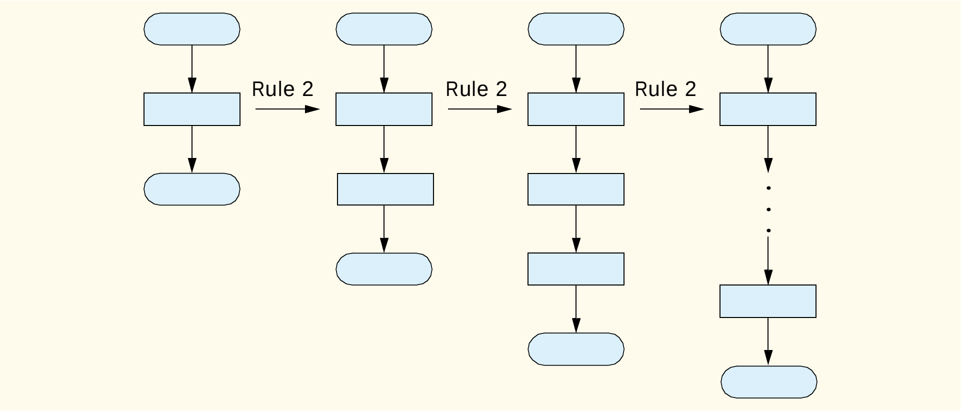

Rule 2
Rule 2
Rule 2

Fig. 5.25 Repeatedly applying rule 2 of Fig. 5.23 to the simplest flowchart.

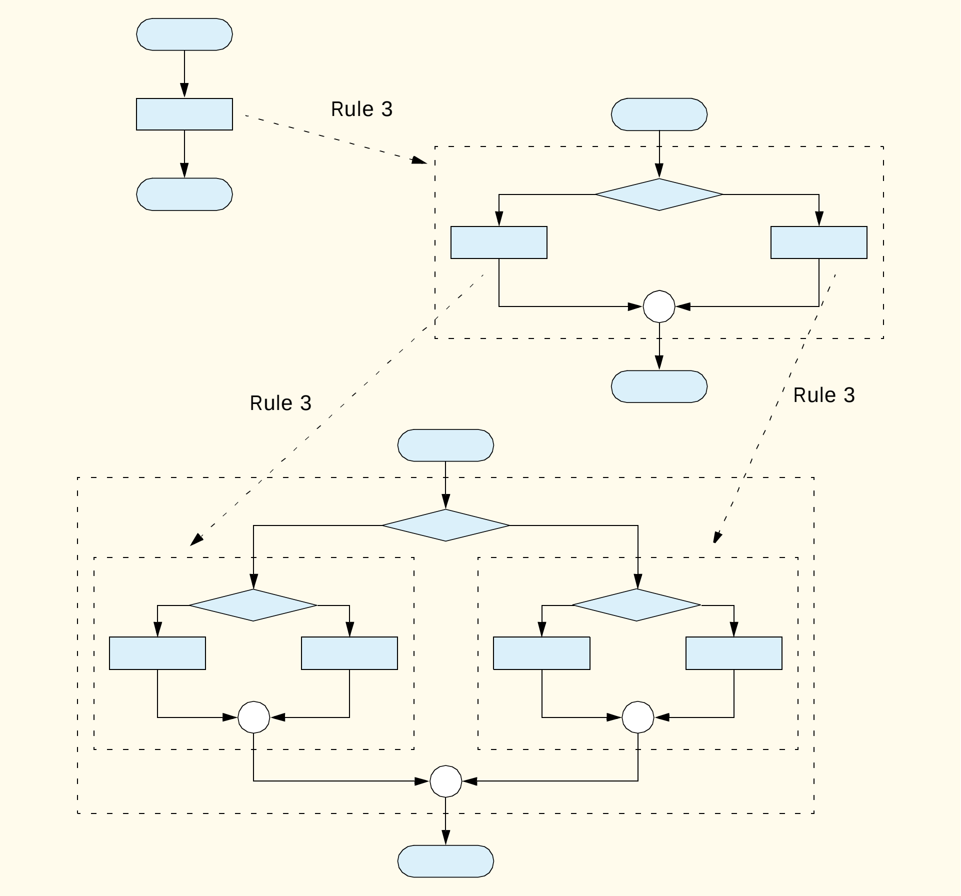

Rule 3
Rule 3
Rule 3

Fig. 5.26 Applying rule 3 of Fig. 5.23 to the simplest flowchart.

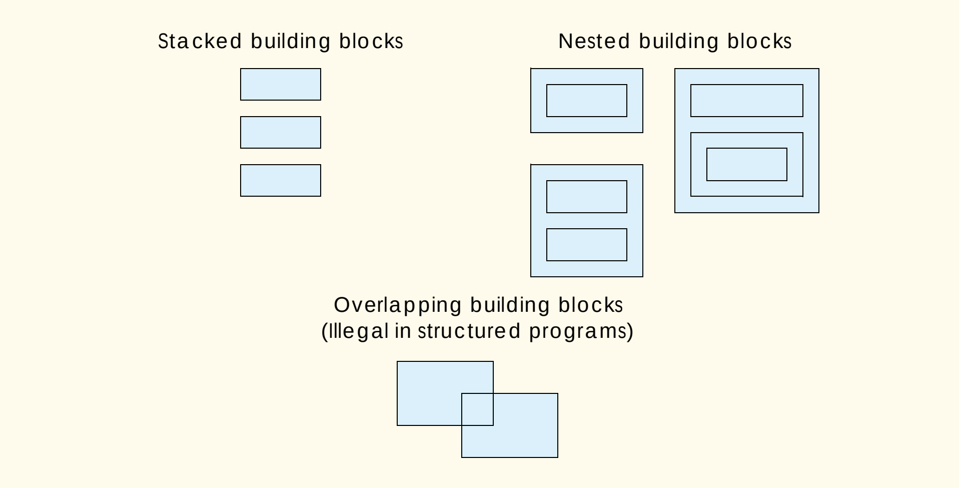

Stacked building blocks
Nested building blocks
Overlapping building blocks
(Illegal in structured programs)

Fig. 5.27 Stacked, nested and overlapped building blocks.

If the rules in Fig. 5.23 are followed, an unstructured flowchart (such as that in Fig. 5.28) cannot be created. If you are uncertain about whether a particular flowchart is structured, apply the rules in Fig. 5.23 in reverse to try to reduce the flowchart to the simplest flowchart. If the flowchart can be reduced to the simplest flowchart, the original flowchart is structured; otherwise, it is not.

In summary, structured programming promotes simplicity. Bohm and Jacopini have found that only three forms of control are necessary:

- Sequence
- Selection
- Repetition

Sequence is trivial. Selection is implemented in one of three ways:

- `if` structure (single selection)
- `if`/`else` structure (double selection)
- `switch` structure (multiple selection)

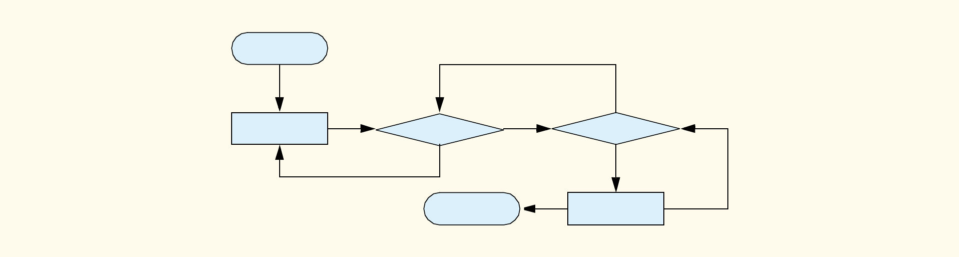


Fig. 5.28 Unstructured flowchart.

In fact, it is straightforward to prove that the `if` structure is sufficient to provide any form of selection. Everything that can be done with the `if`/`else` and `switch` structures can be implemented by combining `if` structures (although perhaps not as elegantly).

Repetition is implemented in one of four ways:

- `while` structure
- `do`/`while` structure
- `for` structure
- `foreach` structure (discussed in Chapter 7)

It is straightforward to prove that the `while` structure is sufficient to provide any form of repetition. Everything that can be done with the `do`/`while`, `for` and `foreach` structures can be done with the `while` structure (although perhaps not as elegantly).

Combining these results illustrates that any form of control ever needed in a C# program can be expressed in terms of:

- sequence
- `if` structure (selection)
- `while` structure (repetition)

These control structures can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

In this chapter, we discussed how to compose programs from control structures that contain actions and decisions. In Chapter 6, Methods, we introduce another program-structuring unit, called the *method*. We will learn to compose large programs by combining methods that are composed of control structures. We also discuss how methods promote software reusability. In Chapter 8, Object-Based Programming, we discuss in more detail another C# program-structuring unit, called the *class*. We then create objects from classes and proceed with our treatment of object-oriented programming—the key focus of this book.

SUMMARY

- Counter-controlled repetition requires the name of a control variable (or loop counter), the initial value of the control variable, the increment (or decrement) by which the control variable is modified each time through the loop and the condition that tests for the final value of the control variable (i.e., whether looping should continue).
- Declarations that include initialization are executable statements.
- Floating-point values may be approximate, so controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate tests for termination.
- If there is more than one statement in the body of the `for`, braces (`{` and `}`) are needed to define the body of the loop.
- Using an incorrect relational operator or an incorrect final value of a loop counter in the condition of a `while`, `for` or `do`/`while` structure can cause an off-by-one error.
- The general format of the `for` structure is

```
for ( expression1; expression2; expression3 )
   statement
```

where *expression1* names the loop's control variable and provides its initial value, *expression2* is the loop-continuation condition (containing the control variable's final value) and *expression3* increments the control variable.

- In most cases, the **`for`** structure can be represented with an equivalent **`while`** structure with *expression1*, *expression2* and *expression3* placed as follows:

```
expression1;

while ( expression2 )
{
   statement
   expression3;
}
```

- If the initialization section in the **`for`** structure header defines the control variable, the control variable can be used only in the body of the **`for`** structure.
- The scope of a variable defines where the variable can be used in a program.
- The three expressions in the **`for`** structure are optional. The two semicolons in the **`for`** structure are required.
- If the loop-continuation condition is initially false, the body of the **`for`** structure does not execute.
- Changing the value of the control variable in the body of a **`for`** loop can lead to subtle errors.
- Do not use variables of type **`float`** or **`double`** to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. Type **`decimal`** is available for performing monetary calculations properly.
- The **`switch`** multiple-selection structure consists of a series of **`case`** labels and an optional **`default`** case. Each label (**`case`** or **`default`**) contains statements to be executed if that label is selected.
- A **`break`** is required in every **`case`** of a **`switch`** structure, except for empty **`case`**s.
- Listing **`case`** labels together (such as **`case 'C': case 'c':`**, with no statements between the cases) causes the same set of actions to be performed for each of the **`case`**s.
- When using the **`switch`** structure, remember that the expression after each **`case`** must be a constant integral expression (i.e., any combination of character and integer constants that evaluates to a constant integer value) or a string.
- The **`do`**/**`while`** structure tests the loop-continuation condition after the loop body executes; therefore, the loop body always executes at least once.
- The **`break`** statement, when executed in a **`while`**, **`for`**, **`do`**/**`while`** or **`switch`** structure, causes immediate exit from that structure. Execution continues with the first statement after the structure.
- The **`continue`** statement, when executed in a **`while`**, **`for`** or **`do`**/**`while`** structure, skips the remaining statements in the body of that structure and proceeds with the next iteration of the loop.
- C# uses conditional and logical operators to form complex conditions by combining simple ones.
- The conditional and logical operators are **`&&`** (conditional AND), **`&`** (logical AND), **`||`** (conditional OR), **`|`** (logical inclusive OR), **`^`** (logical exclusive OR) and **`!`** (logical NOT, also called logical negation).
- The conditional **`&&`** operator ensures that two conditions are both **`true`** before we choose a certain path of execution.
- The logical **`||`** operator ensures that at least one of two conditions is **`true`** before we choose a certain path of execution.

- A condition containing the boolean logical exclusive OR (`^`) operator is true if and only if one of its operands is true and one is false.
- The `!` (logical negation) operator “reverses” the meaning of a condition.
- When a `bool` value is concatenated to a `string`, C# adds the string `"False"` or `"True"` based on the `bool` value.
- In flowcharts, small circles indicate the single entry point and exit point of each structure.
- Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control structures and to build structured programs by properly combining control structures in two simple ways—stacking and nesting.
- Structured programming promotes simplicity.
- Bohm and Jacopini have given us the result that only three forms of control are needed—sequence, selection and repetition.
- Selection is implemented with one of three control structures—`if`, `if`/`else` and `switch`.
- Repetition is implemented with one of four control structures—`while`, `do`/`while`, `for` and `foreach`.
- The `if` structure is sufficient to provide any form of selection.
- The `while` structure is sufficient to provide any form of repetition.

TERMINOLOGY

`!` logical NOT
`!=` ”is not equal to”
`&` logical AND
`&&` conditional AND
`^` boolean logical exclusive OR
`|` boolean logical inclusive OR
`||` conditional OR
`<=` less than or equal
AND operator boolean logical
AND operator logical
binary
binary operator
body of a loop
`bool` values
braces (`{` and `}`)
`break` statement
buttons for message dialogs
`C` formatting code
`case`
conditional AND operator (`&&`)
conditional OR operator (`||`)
`const` variable
constant integral expression
constant variable
`continue` statement
control structure
control variable
controlling expression
control-structure nesting
control-structure stacking
counter variable
counter-controlled repetition
`D` formatting code
`decimal`
decrement expression
`default` statement
delay loop
diamond symbol
`do`/`while` structure
double-selection structure
`E` formatting code
empty `case`
empty statement (semicolon by itself)
entry point of a control structure
`Error`
`F` formatting code
flowchart symbol
`for` structure
`for` structure header
`foreach` structure
formatting code
formatting data
`G` formatting code
`goto` statement

hexadecimal (base16) number system
icons for message dialogs
`if` structure
`if`/`else` structure
increment expression
infinite loop
initialization section of a `for` structure
iteration of a loop
labels in a `switch` structure
levels of nesting
logical AND operator (`&`)
logical exclusive OR operator (^)
logical negation (`!`)
logical NOT or logical negation operator (`!`)
logical operators
logical OR operator (|)
loop body
loop counter
loop-continuation condition
`Math` class
message-dialog buttons
message-dialog icons
`MessageBoxButton.AbortRetryIgnore`
`MessageBoxButton.OK`
`MessageBoxButton.OKCancel`
`MessageBoxButton.RetryCancel`
`MessageBoxButton.YesNo`
`MessageBoxButton.YesNoCancel`
`MessageBoxIcon.Error`
`MessageBoxIcon.Exclamation`
`MessageBoxIcon.Information`
`MessageBoxIcon.Question`
multiple-selection structure
`N` formatting code
nested building block
nested control structure
nesting rule
off-by-one error
one-based counting
optimization
overlapped building block
`Pow` method of class `Math`
program-construction principles
rectangle symbol
repetition
scope of a variable
selection
sequence
short-circuit evaluation
side effect
simple condition
simplest flowchart
single selection
single-entry/single-exit control structure
small circle symbol
stacking
stacking rule
`string` formatting codes
structured programming
`switch` structure
title bar string
truth table
unary operator
unstructured flowchart
`while` structure
`X` formatting code
zero-based counting

SELF-REVIEW EXERCISES

5.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `default` case is required in the `switch` selection structure.

b) If there is more than one statement in the body of the `for`, braces (`{` and `}`) are needed to define the body of the loop.

c) The expression `( x > y && a < b )` is true if either `x > y` is true or `a < b` is true.

d) An expression containing the || operator is true if either or both of its operands is true.

e) The expression `( x <= y && y > 4 )` is true if `x` is less than or equal to `y` or `y` is greater than 4.

f) A `for` loop requires two commas in its header.

g) Infinite loops are caused when the loop-termination condition is always true.

h) The following syntax continues iterating the loop while $10 < x < 100$:

```
while ( x > 10 && x < 100 );
```

i) The **`break`** statement, when executed in a repetition structure, causes immediate exit from the repetition structure.

j) The `||` operator has a higher precedence than the **`&&`** operator.

5.2 Fill in the blanks in each of the following statements:

a) Specifying the order in which statements are to be executed in a computer program is called __________.

b) Placing a semicolon after a **`for`** statement typically results in a __________ error.

c) A **`for`** loop should count with __________ values.

d) Using the **`<`** relational operator instead of **`<=`** in a **`while`**-repetition condition that should loop 10 times (as shown below) causes an __________ error:

```
int x = 1;
while ( x < 10 ) …
```

e) A control variable initialized within a **`for`** loop can be used only in the body of the loop. This is called the __________ of the variable.

f) In a **`for`** loop, incrementing occurs __________ the body of the structure is performed each time.

g) Multiple initializations in the **`for`** structure header should be separated by __________.

h) Placing expressions whose values do not change inside __________ can lead to poor performance.

i) The four types of **`MessageBox`** icons are exclamation, information, error and __________.

j) The value in parentheses immediately following the keyword **`switch`** is called the __________.

5.3 Write a C# statement or a set of C# statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99, using a **`for`** structure. Assume that the integer variables **`sum`** and **`count`** have been declared.

b) Calculate the value of **`2.5`** raised to the power of **`3`**, using the **`Math.Pow`** method.

c) Print the integers from 1 to 20, using a **`while`** loop and the counter variable **`x`**. Assume that the variable **`x`** has been declared, but not initialized. Print only five integers per line. [*Hint*: Use the calculation **`x % 5`**. When the value of this is 0, print a newline character; otherwise, print a tab character. Use the **`Console.WriteLine()`** method to output the newline character, and use the **`Console.Write( '\t' )`** method to output the tab character.]

d) Repeat part c, using a **`for`** structure.

ANSWERS TO SELF-REVIEW EXERCISES

5.1 a) False. The **`default`** case is optional. If no default action is required, then there is no need for a **`default`** case. b) True. c) False. Both of the relational expressions must be true for the entire expression to be true. d) True. e) False. The expression **`( x <= y && y > 4 )`** is true if **`x`** is less than or equal to **`y`** and **`y`** is greater than **`4`**. f) False. A **`for`** loop requires two semicolons in its header. g) False. Infinite loops are caused when the loop-continuation condition is always true. h) True. i) True. j) False. The **`&&`** operator has higher precedence than the `||` operator.

5.2 a) program control. b) logic. c) integral. d) off-by-one. e) scope. f) after. g) comma. h) loops. i) question. j) controlling expression.

5.3 a)

```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
   sum += count;
```

b) `Math.Pow( 2.5, 3 )`

c)
```
x = 1;

while ( x <= 20 )
{
   Console.Write( x );

   if ( x % 5 == 0 )
      Console.WriteLine();
   else
      Console.Write( '\t' );

   ++x;
}
```

d)
```
for ( x = 1; x <= 20; x++ )
{
   Console.Write( x );

   if ( x % 5 == 0 )
      Console.WriteLine();
   else
      Console.Write( '\t' );
}
```

or

```
for ( x = 1; x <= 20; x++ )

   if ( x % 5 == 0 )
      Console.WriteLine( x );
   else
      Console.Write( x + "\t" );
```

EXERCISES

5.4 The *factorial* method is used frequently in probability problems. The factorial of a positive integer *n* (written *n!* and pronounced “n factorial”) is equal to the product of the positive integers from 1 to *n*. Write a program that evaluates the factorials of the integers from 1 to 20 with different integer data types. Display the results in a three-column output table. [*Hint*: create a Windows application, using `Label`s as the columns and the `'\n'` character to line up rows.] The first column should display the *n* values (1-20). The second column should display *n!*, calculated with `int` (`Int32`, a 32-bit integer value). The third column should display *n!*, calculated with `long` (`Int64`, a 64-bit integer value). What happens when `int` (`Int32`) is too small in size to hold the result of a factorial calculation?

5.5 Write two programs that each print a table of the binary, octal, and hexadecimal equivalents of the decimal numbers in the range 1–256. If you are not familiar with these number systems, read Appendix C, Number Systems, first.

a) For the first program, print the results to the console without using any `string` formats.
b) For the second program, print the results to the console using both the decimal and hexadecimal `string` formats (there are no formats for binary and octal in C#).

5.6 (*Pythagorean Triples*) A right triangle can have sides that are all integers. A set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of the two sides is equal to the square of the hypotenuse. Write a program to find all Pythagorean triples for `side1`, `side2` and `hypotenuse`, none larger than 30. Use a triple-nested `for` loop that tries all possibilities. This is an example of "brute force" computing. You will learn in more advanced computer science courses that there are several problems for which there is no other known algorithmic approach.

5.7 Write a program that displays the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (`*`) should be printed by a single statement of the form `Console.Write( '*' );` (this causes the asterisks to print side by side). A statement of the form `Console.WriteLine();` can be used to position to the next line. A statement of the form `Console.Write( ' ' );` can be used to display spaces for the last two patterns. There should be no other output statements in the program. [*Hint*: The last two patterns require that each line begin with an appropriate number of blanks.]

```
(A)         (B)         (C)         (D)

*           **********  **********           *
**          *********    *********          **
***         ********      ********         ***
****        *******        *******        ****
*****       ******          ******       *****
******      *****            *****      ******
*******     ****              ****     *******
********    ***                ***    ********
*********   **                  **   *********
**********  *                    *  **********
```

5.8 Modify Exercise 5.7 to combine your code from the four separate triangles of asterisks into a single program that prints all four patterns side by side, making clever use of nested `for` loops.

5.9 Write a program that prints the following diamond shape. You may use output statements that print a single asterisk (`*`), a single space or a single newline character. Maximize your use of repetition (with nested `for` structures) and minimize the number of output statements.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.10 Modify the program you wrote in Exercise 5.9 to read an odd number in the range from 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

6

Methods

Objectives

- To construct programs modularly from small pieces called methods.
- To introduce the common math methods available in the Framework Class Library.
- To be able to create new methods.
- To understand the mechanisms for passing information between methods.
- To introduce simulation techniques that use random number generation.
- To understand how the visibility of identifiers is limited to specific regions of programs.
- To understand how to write and use methods that call themselves.

Form ever follows function.
Louis Henri Sullivan

E pluribus unum.
(One composed of many.)
Virgil

O! call back yesterday, bid time return.
William Shakespeare

Call me Ishmael.
Herman Melville

When you call me that, smile.
Owen Wister

Outline

6.1 **Introduction**
6.2 **Program Modules in C#**
6.3 **`Math` Class Methods**
6.4 **Methods**
6.5 **Method Definitions**
6.6 **Argument Promotion**
6.7 **C# Namespaces**
6.8 **Value Types and Reference Types**
6.9 **Passing Arguments: Pass-by-Value vs. Pass-by-Reference**
6.10 **Random-Number Generation**
6.11 **Example: Game of Chance**
6.12 **Duration of Variables**
6.13 **Scope Rules**
6.14 **Recursion**
6.15 **Example Using Recursion: The Fibonacci Series**
6.16 **Recursion vs. Iteration**
6.17 **Method Overloading**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

6.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters of this text. Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or *modules*. This technique is known as *divide and conquer*. This chapter describes many key features of the C# language that facilitate the design, implementation, operation and maintenance of large programs.

6.2 Program Modules in C#[1]

Modules in C# are called *methods* and *classes*. C# programs are written by combining new methods and classes that the programmer writes with "prepackaged" methods and classes available in the *.NET Framework Class Library* (*FCL*). In this chapter, we concentrate on methods. We discuss classes in detail in Chapter 8, Object-Based Programming.

The FCL provides a rich collection of classes and methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, error checking and many other useful operations. This set of modules makes the programmer's job easier, because the modules provide many of the capabilities programmers need. The FCL methods are part of the .NET Framework, which includes FCL classes **`Console`** and **`MessageBox`** used in earlier examples.

---

[1] It is important to note that we are discussing modules in an abstract sense. In C#, there is another form of code packaging (other than assemblies), called modules. This is not what we are discussing in this chapter, but the reader should know that this term can be used in two ways.

Software Engineering Observation 6.1

Familiarize yourself with the rich collection of classes and methods in the FCL.

Software Engineering Observation 6.2

When possible, use .NET Framework classes and methods instead of writing new classes and methods. This reduces program development time and avoids the introduction of new errors.

The programmer can write methods to define specific tasks that may be used at many points in a program. Such methods are known as *programmer-defined* (or *user-defined*) *methods*. The actual statements defining the method are written only once and are hidden from other methods.

A method is *invoked* (i.e., made to perform its designated task) by a *method call*. The method call specifies the name of the method and may provide information (as *arguments*) that the called method requires to perform its task. When the method call completes, the method either returns a result to the *calling method* (or *caller*) or simply returns control to the calling method. A common analogy for this is the hierarchical form of management. A boss (the calling method or caller) asks a worker (the *called method*) to perform a task and report back (i.e., *return*) the results after completing the task. The boss method does not know *how* the worker method performs its designated tasks. The worker may also call other worker methods, and the boss will be unaware of these calls. We will see how this "hiding" of implementation details promotes good software engineering. Figure 6.1 shows a **`boss`** method communicating with worker methods **`worker1`**, **`worker2`** and **`worker3`** in a hierarchical manner. Note that **`worker1`** acts as a "boss" method to **`worker4`** and **`worker5`** in this particular example.

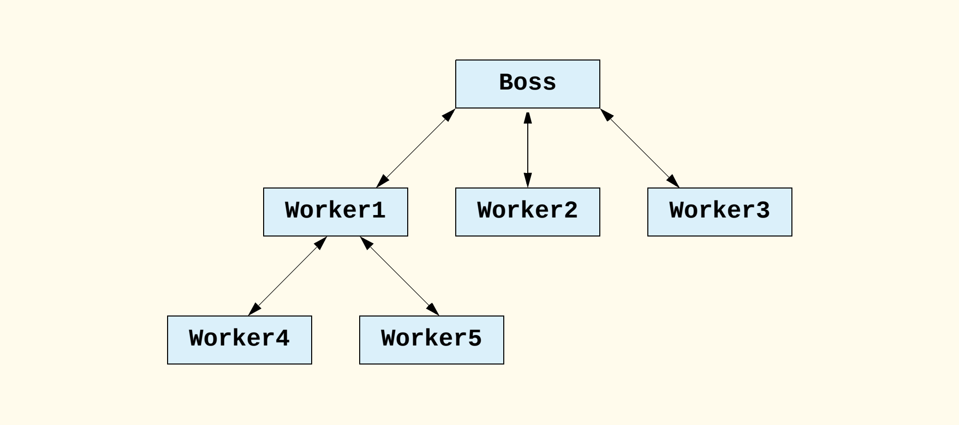


Fig. 6.1 Hierarchical boss method/worker method relationship.

6.3 `Math` Class Methods

`Math` class methods allow the programmer to perform certain common mathematical calculations. We use various `Math` class methods to introduce the concept of methods in general. Throughout the book, we discuss many other methods from the classes of the Framework Class Library.

Methods are called by writing the name of the method, followed by a left parenthesis, the *argument* (or a comma-separated list of arguments) of the method and a right parenthesis. The parentheses may be empty, if we are calling a method that needs no information to perform its task. For example, a programmer wishing to calculate and print the square root of `900.0` might write

```
Console.WriteLine( Math.Sqrt( 900.0 ) );
```

When this statement executes, the method `Math.Sqrt` calculates the square root of the number in parentheses (`900.0`). The number `900.0` is the argument to the `Math.Sqrt` method. The `Math.Sqrt` method takes an argument of type `double` and returns a result of type `double`. The preceding statement uses the result of method `Math.Sqrt` as the argument to method `Console.WriteLine` and displays `30.0`. Note that all `Math` class methods must be invoked by preceding the method name with the class name `Math` and a dot (`.`) operator (also called the member access operator).

Software Engineering Observation 6.3

It is not necessary to add an assembly reference to use the `Math` class methods in a program. Class `Math` is located in namespace `System`, which is available to every program.

Common Programming Error 6.1

Forgetting to invoke a `Math` class method by preceding the method name with the class name `Math` and a dot operator (`.`) results in a syntax error.

Method arguments may be constants, variables or expressions. If `c1 = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

```
Console.WriteLine( Math.Sqrt( c1 + d * f ) );
```

calculates and displays the square root of `13.0 + 3.0 * 4.0 = 25.0`, which is `5.0`.

Figure 6.2 summarizes some `Math` class methods. In this figure, the variables `x` and `y` are of type `double`; however, many of the methods provide versions that take values of other data types as arguments. The `Math` class also defines two commonly used mathematical constants—`Math.PI` (3.14159265358979323846) and `Math.E` (2.7182818284590452354). The constant `Math.PI` of class `Math` is the ratio of a circle's circumference to its diameter. The constant `Math.E` is the base value for natural logarithms (calculated with the `Math.Log` method).

6.4 Methods

Methods allow programmers to modularize programs. Variables declared in method definitions are *local variables*—only the method that defines them knows they exist. Most methods have a list of *parameters* that enable method calls to communicate information between methods. A method's parameters are also variables local to that method and are not visible in any other methods.

Method	Description	Example

`Abs( x )`	absolute value of *x*	`Abs( 23.7 )` is `23.7`
		`Abs( 0 )` is `0`
		`Abs( -23.7 )` is `23.7`

`Ceiling( x )`	rounds *x* to the smallest integer not less than *x*	`Ceiling( 9.2 )` is `10.0` `Ceiling( -9.8 )` is `-9.0`
`Cos( x )`	trigonometric cosine of *x* (*x* in radians)	`Cos( 0.0 )` is `1.0`
`Exp( x )`	exponential method e^x	`Exp( 1.0 )` is approximately `2.7182818284590451` `Exp( 2.0 )` is approximately `7.3890560989306504`
`Floor( x )`	rounds *x* to the largest integer not greater than *x*	`Floor( 9.2 )` is `9.0` `Floor( -9.8 )` is `-10.0`
`Log( x )`	natural logarithm of *x* (base *e*)	`Log( 2.7182818284590451 )` is approximately `1.0` `Log( 7.3890560989306504 )` is approximately `2.0`
`Max( x, y )`	larger value of *x* and *y* (also has versions for `float`, `int` and `long` values)	`Max( 2.3, 12.7 )` is `12.7` `Max( -2.3, -12.7 )` is `-2.3`
`Min( x, y )`	smaller value of *x* and *y* (also has versions for `float`, `int` and `long` values)	`Min( 2.3, 12.7 )` is `2.3` `Min( -2.3, -12.7 )` is `-12.7`
`Pow( x, y )`	*x* raised to power *y* (x^y)	`Pow( 2.0, 7.0 )` is `128.0` `Pow( 9.0, .5 )` is `3.0`
`Sin( x )`	trigonometric sine of *x* (*x* in radians)	`Sin( 0.0 )` is `0.0`
`Sqrt( x )`	square root of *x*	`Sqrt( 900.0 )` is `30.0` `Sqrt( 9.0 )` is `3.0`
`Tan( x )`	trigonometric tangent of *x* (*x* in radians)	`Tan( 0.0 )` is `0.0`

Fig. 6.2 Commonly used `Math` class methods.

There are several motivations for modularizing a program with methods. The divide-and-conquer approach makes program development more manageable. Another motivation is *software reusability*—using existing methods (and classes) as building blocks to create new programs. With proper method naming and definition, we can create programs from standardized methods, rather than building customized code. For example, we did not have to define how to convert `string`s to integers—The .NET Framework Class Library already defines such methods for us (`Int32.Parse`). A third motivation is to avoid repeating code in a program. Packaging code as a method allows that code to be executed from several locations in a program—we simply have to call that method.

Good Programming Practice 6.1

Make good use of modularity to increase the clarity and organization of your program. This will not only help others understand your program, but it also will aid in program development, testing and debugging.

Software Engineering Observation 6.4

To promote reusability, each method should perform a single, well-defined task, and the name of the method should express that task effectively.

Software Engineering Observation 6.5

If you cannot choose a concise name that expresses what the method does, it is possible that your method is attempting to perform too many diverse tasks. Usually it is best to break such a method into several smaller methods.

6.5 Method Definitions

The programs presented up to this point each contained at least one method definition (such as **`Main`**) that called FCL methods to accomplish the program's tasks. We now consider how to write customized methods.

Consider the Windows application in Fig. 6.3, which uses a method called **`Square`** to calculate the squares of the integers from 1 to 10. Notice the comment on line 15

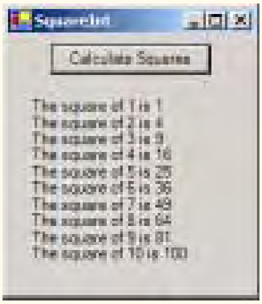
```
// Visual Studio .NET generated code
```

Throughout the book we will use this comment to denote code that we are not displaying in the chapter. In all cases, this represents code created by the IDE. Most of this code initializes properties of GUI components. The examples in all chapters of the book are included in their entirety on the CD that accompanies this book. The examples also can be downloaded from **`www.deitel.com`** via the **Downloads/Resources** link. Figure 6.4, displays all the code. However, you do not need to understand all the code at this point in the book. Chapter 12, Graphical User Interfaces: Part 1, presents an example in which we discuss all the code in detail.

```
 1  // Fig. 6.3: SquareInt.cs
 2  // Demonstrates a programmer-defined Square method.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.Data;
 9
10  public class SquareInt : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.Button calculateButton;
13     private System.Windows.Forms.Label outputLabel;
14
15     // Visual Studio .NET generated code
```

Fig. 6.3 Using programmer-defined method **`Square`**. (Part 1 of 2.)

```

   [STAThread]
   static void Main()
   {
      Application.Run( new SquareInt() );
   }

   // Square method definition
   int Square( int y )
   {
      return y * y; // return square of y

   } // end method Square

   private void calculateButton_Click( object sender,
      System.EventArgs e )
   {
      outputLabel.Text = "";

      // loop 10 times
      for ( int counter = 1; counter <= 10; counter++ )
      {
         // calculate square of counter and store in result
         int result = Square( counter );

         // append result to outputLabel
         outputLabel.Text += "The square of " + counter +
            " is " + result + "\n";
      }

   } // end method calculateButton_Click

} // end of class SquareInt
```



```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
```

Fig. 6.3 Using programmer-defined method **`Square`**. (Part 2 of 2.)

Until now, our programs have used methods of class **`Console`** to obtain user input from the command prompt. These programs output their results either to the command prompt or in **`MessageBox`**es. Although these are valid ways to receive input from a user and display output, they are fairly limited in their capabilities—the command prompt can obtain only one value at a time from the user, and a message dialog can display only one message. It is much more common for programs to read multiple inputs simultaneously (e.g., when the user enters name and address information) or to display many pieces of data at once (such as the values of the first ten squares in this example). To introduce more elaborate user interfaces, the program in Fig. 6.3 illustrates two graphical user interface concepts—attaching multiple GUI components to an application and *event handling*.

To build this application, we use the Windows Form Designer to add a `Button` GUI component object to the `Form`. This example also uses a label to display the results of calculating the first ten squares. The program invokes a special method, known as an *event handler*, when the user clicks the **Calculate Squares** button. An event handler is a method that performs some action in response to an *event*. Events occur when certain actions take place in a graphical user interface, such as when the user clicks a button. Using GUI component objects and events together allows programmers to create applications that interact with users in more sophisticated ways than we have seen previously. In Visual Studio .NET's Windows Form Designer, double clicking on a GUI component object causes Visual Studio .NET to generate an empty event handler method. The event handler method's name defaults to the GUI component's name, followed by an underscore and the name of the event. The programmer then can fill in the event handler method with code that performs a particular task.

In Fig. 6.3, method `calculateButton_Click` (lines 30–46) is an event handler method for `calculateButton`'s `Click` event (i.e., the event that occurs when the user clicks the button). When the user clicks the button, line 33 assigns the empty string (`""`) to `outputLabel`'s `Text` property to ensure that the output does not scroll off the form if the user presses the **Calculate Squares** button more than once. Lines 36–44 repeatedly invoke method `Square` to calculate the squares of the integers from 1 to 10. Line 39 invokes the `Square` method and passes the variable `counter` as an argument.

As in some of our previous Windows applications, we create a label called `outputLabel` to display the program's output. Every label contains a `string` property called `Text`, which can be accessed using the dot operator (`.`). We append the results of the square calculations to this label's `Text` property.

Line 39 declares `int` variable `result` to store the result of each square calculation. Lines 36–44 contain a `for` repetition structure in which each iteration of the loop calculates the `Square` of the current value of control variable `counter` and stores the value in `result`. Lines 42–43 concatenate each result to the `Text` property of `outputLabel`. At the end of the loop, the `Label` contains the results of squaring the values from 1 to 10.

The program invokes method `Square` on line 39. The parentheses, `()`, after `Square` represent the *method-call operator*, which has high precedence. At this point, the program makes a copy of the value of `counter` (the argument to the method call), and program control transfers to method `Square` (defined at lines 24–28). Method `Square` receives the copy of the value of `counter` in the *parameter* `y`. Then, `Square` calculates `y * y` (line 26). Method `Square` uses a ***`return`*** statement to return (i.e., give back) the result of the calculation to the statement that invoked `Square` (located in line 39). Line 39 then assigns the returned value to variable `result`. Lines 42–43 concatenate `"The square of"`, the value of `counter`, `" is "`, the value of `result` and a newline character to the end of `outputLabel`'s `Text` property. The `for` repetition structure repeats this process 10 times.

The definition of method `Square` (line 24) shows (inside the parentheses) that `Square` expects an integer parameter `y`. Parameter `y` is the name that holds the value passed to `Square` as an argument. The parameter name provides access to the argument value, so that code in the method body can use the value. Keyword `int`, which precedes the method name, indicates that method `Square` returns an integer result. The `return` statement in `Square` (line 26) passes the result of the calculation `y * y` back to the calling statement. Note that the entire method definition appears inside the braces of class `SquareInt`. All methods must be defined inside a class definition.

Good Programming Practice 6.2

Place a blank line between method definitions to separate the methods and enhance program readability.

Common Programming Error 6.2

Defining a method outside the braces of a class definition is a syntax error.

The format of a method definition is

return-value-type *method-name* `(` *parameter-list* `)`
`{`
declarations and statements
`}`

The first line is sometimes known as the *method header*. The *method-name* is any valid identifier. The *return-value-type* is the data type of the result that the method returns to its caller. The *return-value-type* `void` indicates that a method does not return a value. Methods can return at most one value.

Common Programming Error 6.3

Omitting the return-value-type *in a method definition is a syntax error. If a method does not return a value, the method's* return-value-type *must be* `void`*.*

Common Programming Error 6.4

Forgetting to return a value from a method that is supposed to return a value is a syntax error. If a return-value-type *other than* `void` *is specified, the method must contain a* `return` *statement that returns a value.*

Common Programming Error 6.5

Returning a value from a method whose return type has been declared `void` *is a syntax error.*

The *parameter-list* is a comma-separated list in which the method declares each parameter's type and name. The method call must specify one argument for each parameter in the method definition and the arguments must appear in the same order as the parameters in the method definition. The arguments also must be compatible with the parameter's type. For example, a parameter of type `double` could receive values of 7.35, 22 or –.03546, but not `"hello"` because a `double` variable cannot contain a `string`. If a method does not receive any values, the parameter list is empty (i.e., the method name is followed by an empty set of parentheses). Each parameter in a method's parameter list must have a data type; otherwise, a syntax error occurs.

Common Programming Error 6.6

Declaring method parameters of the same type as `float x, y` *instead of* `float x, float y` *is a syntax error, because types are required for each parameter in the parameter list.*

Common Programming Error 6.7

Placing a semicolon after the right parenthesis enclosing the parameter list of a method definition is a syntax error.

Common Programming Error 6.8

Redefining a method parameter in the method's body is a syntax error.

Common Programming Error 6.9

Passing to a method an argument that is not compatible with the corresponding parameter's type is a syntax error.

The declarations and statements within braces form the *method body*. The method body is also referred to as a block. As discussed previously, a block is a set of declarations and statements enclosed in curly braces. Variables can be declared in any block, and blocks can be nested.

Common Programming Error 6.10

Defining a method inside another method is a syntax error (i.e., methods cannot be nested).

Good Programming Practice 6.3

Choosing meaningful method names and parameter names makes programs more readable and helps avoid excessive use of comments.

Software Engineering Observation 6.6

As a rule of thumb, a method should be no longer than one page. Better yet, a method should be no longer than half a page. Regardless of how long a method is, it should perform one task well. Small methods promote software reusability.

Testing and Debugging Tip 6.1

Small methods are easier to test, debug and understand than large methods.

Software Engineering Observation 6.7

A method requiring a large number of parameters may be performing too many tasks. Consider dividing the method into smaller methods that perform separate tasks. As a rule of thumb, the method header should fit on one line (if possible).

Software Engineering Observation 6.8

The number, type and order of arguments in a method call must exactly match those of the parameters in the corresponding method header.

There are three ways to return control to the point at which a method was invoked. If the method does not return a result (i.e., the method has a `void` return type), control returns when the program reaches the method-ending right brace or when the statement

```
return;
```

executes. If the method does return a result, the statement

```
return expression;
```

returns the value of *expression* to the caller. When a `return` statement executes, control returns immediately to the point at which the method was invoked.

Notice the syntax that invokes method `Square` in Fig. 6.3—we use the method name, followed by the arguments to the method in parentheses. Methods in a class definition are allowed to invoke all other methods in the same class definition by using this syntax (an exception to this is discussed in Chapter 8, Object-Based Programming). We now have seen three ways to call a method—a method name by itself (as shown with `Square( x )`), a reference to an object followed by the dot (`.`) operator and the method name (such as `string1.CompareTo(string2)`) and a class name followed by a method name (such as `Math.Sqrt( 9.0 )`). The last syntax is for calling the ***`static`*** *methods* of a class (discussed in detail in Chapter 8, Object-Based Programming).

The application in our next example (Fig. 6.4) uses programmer-defined method `Maximum` to determine and return the largest of three floating-point values that the user inputs through the program's graphical user interface. Note that in this example, we show all of the code that the Windows Form Designer generates. Throughout the rest of the book, we omit portions of the generated code that are not relevant to our discussions. In such programs, we place a comment that indicates where the Visual Studio .NET generated code appears in the original source file.

```
// Fig. 6.4: MaximumValue.cs
// Finding the maximum of three double values.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace MaximumValue
{
   /// <summary>
   /// Summary description for Form1.
   /// </summary>
   public class MaximumValue : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Label firstNumberLabel;
      private System.Windows.Forms.Label secondNumberLabel;
      private System.Windows.Forms.Label thirdNumberLabel;
      private System.Windows.Forms.Label maximumLabel;
      private System.Windows.Forms.TextBox firstNumberTextBox;
      private System.Windows.Forms.TextBox secondNumberTextBox;
      private System.Windows.Forms.TextBox thirdNumberTextBox;
      private System.Windows.Forms.Button calculateButton;

      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components = null;

```

Fig. 6.4 Programmer-defined **`Maximum`** method. (Part 1 of 5.)

```
      public MaximumValue()
      {
         //
         // Required for Windows Form Designer support
         //
         InitializeComponent();

         //
         // TODO: Add any constructor code after
         //       InitializeComponent call
         //
      }

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose( bool disposing )
      {
         if( disposing )
         {
            if (components != null)
            {
               components.Dispose();
            }
         }
         base.Dispose( disposing );
      }

      #region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.calculateButton =
            new System.Windows.Forms.Button();
         this.secondNumberTextBox =
            new System.Windows.Forms.TextBox();
         this.thirdNumberTextBox =
            new System.Windows.Forms.TextBox();
         this.firstNumberLabel =
            new System.Windows.Forms.Label();
         this.secondNumberLabel =
            new System.Windows.Forms.Label();
         this.thirdNumberLabel =
            new System.Windows.Forms.Label();
         this.maximumLabel = new System.Windows.Forms.Label();
         this.firstNumberTextBox =
            new System.Windows.Forms.TextBox();
         this.SuspendLayout();
```

Fig. 6.4 Programmer-defined **`Maximum`** method. (Part 2 of 5.)

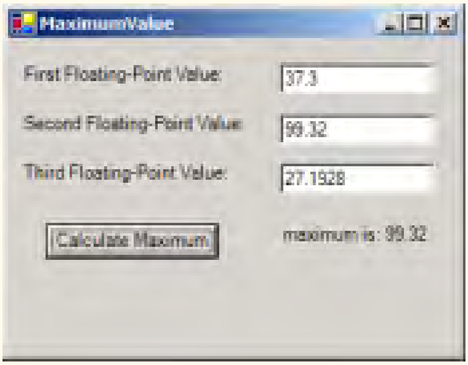
```
            //
            // calculateButton
            //
            this.calculateButton.Location =
               new System.Drawing.Point(24, 120);
            this.calculateButton.Name = "calculateButton";
            this.calculateButton.Size =
               new System.Drawing.Size(112, 23);
            this.calculateButton.TabIndex = 0;
            this.calculateButton.Text = "Calculate Maximum";
            this.calculateButton.Click +=
               new System.EventHandler(this.calculateButton_Click);
            //
            // secondNumberTextBox
            //
            this.secondNumberTextBox.Location =
               new System.Drawing.Point(176, 49);
            this.secondNumberTextBox.Name = "secondNumberTextBox";
            this.secondNumberTextBox.TabIndex = 2;
            this.secondNumberTextBox.Text = "";
            //
            // thirdNumberTextBox
            //
            this.thirdNumberTextBox.Location =
               new System.Drawing.Point(176, 81);
            this.thirdNumberTextBox.Name = "thirdNumberTextBox";
            this.thirdNumberTextBox.TabIndex = 3;
            this.thirdNumberTextBox.Text = "";
            //
            // firstNumberLabel
            //
            this.firstNumberLabel.Location =
               new System.Drawing.Point(8, 16);
            this.firstNumberLabel.Name = "firstNumberLabel";
            this.firstNumberLabel.Size =
               new System.Drawing.Size(136, 23);
            this.firstNumberLabel.TabIndex = 4;
            this.firstNumberLabel.Text =
               "First Floating-Point Value:";
            //
            // secondNumberLabel
            //
            this.secondNumberLabel.Location =
               new System.Drawing.Point(8, 48);
            this.secondNumberLabel.Name = "secondNumberLabel";
            this.secondNumberLabel.Size =
               new System.Drawing.Size(152, 23);
            this.secondNumberLabel.TabIndex = 5;
            this.secondNumberLabel.Text =
               "Second Floating-Point Value:";
            //
            // thirdNumberLabel
            //
```

Fig. 6.4 Programmer-defined `Maximum` method. (Part 3 of 5.)

```
         this.thirdNumberLabel.Location =
            new System.Drawing.Point(8, 80);
         this.thirdNumberLabel.Name = "thirdNumberLabel";
         this.thirdNumberLabel.Size =
            new System.Drawing.Size(144, 23);
         this.thirdNumberLabel.TabIndex = 6;
         this.thirdNumberLabel.Text =
            "Third Floating-Point Value:";
         //
         // maximumLabel
         //
         this.maximumLabel.Location =
            new System.Drawing.Point(176, 120);
         this.maximumLabel.Name = "maximumLabel";
         this.maximumLabel.Size =
            new System.Drawing.Size(100, 80);
         this.maximumLabel.TabIndex = 7;
         //
         // firstNumberTextBox
         //
         this.firstNumberTextBox.Location =
            new System.Drawing.Point(176, 16);
         this.firstNumberTextBox.Name = "firstNumberTextBox";
         this.firstNumberTextBox.TabIndex = 1;
         this.firstNumberTextBox.Text = "";
         //
         // MaximumValue
         //
         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
         this.ClientSize = new System.Drawing.Size(292, 205);
         this.Controls.AddRange(
            new System.Windows.Forms.Control[] {
               this.firstNumberTextBox,
               this.maximumLabel,
               this.thirdNumberLabel,
               this.secondNumberLabel,
               this.firstNumberLabel,
               this.thirdNumberTextBox,
               this.secondNumberTextBox,
               this.calculateButton
            }
         );
         this.Name = "MaximumValue";
         this.Text = "MaximumValue";
         this.ResumeLayout(false);

      }
      #endregion

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
```

Fig. 6.4 Programmer-defined **`Maximum`** method. (Part 4 of 5.)

```
      static void Main()
      {
         Application.Run(new MaximumValue());
      }

      // Method Maximum uses method Math.Max to determine the
      // maximum value among the three double arguments
      double Maximum( double x, double y, double z )
      {
         return Math.Max( x, Math.Max( y, z ) );
      }

      // get the floating-point values that the user entered and
      // invoke method Maximum to determine the maximum value
      private void calculateButton_Click( object sender,
         System.EventArgs e )
      {
         // get inputted values and convert strings to doubles
         double number1 =
            Double.Parse( firstNumberTextBox.Text );

         double number2 =
            Double.Parse( secondNumberTextBox.Text );

         double number3 =
            Double.Parse( thirdNumberTextBox.Text );

         // invoke method Maximum to determine the largest value
         double maximum = Maximum( number1, number2, number3 );

         // display maximum value
         maximumLabel.Text = "maximum is: " + maximum;

      }  // end method calculateButton_Click

   } // end class MaximumValue

} // end namespace MaximumValue
```



First Floating-Point Value: 37.3

Second Floating-Point Value: 99.32

Third Floating-Point Value: 27.1928

maximum is: 99.32

Fig. 6.4 Programmer-defined `Maximum` method. (Part 5 of 5.)

The graphical user interface for this program consists of three **`TextBox`**es in which the user can enter floating-point numbers, a **`Button`** for calculating the maximum, **`Label`**s for each **`TextBox`** and a **`Label`** for displaying the maximum value. Lines 31–182 contain the Visual Studio .NET generated code for constructing this graphical user interface. Lines 31–42 define a special type of method called a *constructor*. Programs invoke constructors to create objects. The constructor performs tasks necessary for preparing an object for use in a program. We discuss constructors in detail in Chapter 8. In the case of Windows applications, the constructor invokes method **`InitializeComponent`** to create the program's graphical user interface (line 36). Method **`InitializeComponent`** (lines 64–181) configures and arranges the program's graphical user interface component objects, such as its **`Label`**s, **`Button`**s and **`TextBox`**es.

When the user closes a program's window, the system invokes method **`Dispose`** (lines 47–57) to "clean up" resources used by the Window.

To create the graphical user interface for this program, drag the appropriate components from the **Toolbox** onto the **`Form`** in the Windows Form Designer. Arrange the components as shown in the screen capture of Fig. 6.4 and set the **`Text`** properties for the **`Label`**s and **`Button`**. Then, double click the **Calculate Maximum** button to add an empty event handler. Fill in this empty event handler with the code shown on lines 202–221. Lines 206–213 invoke **`Double`** method **`Parse`** on the **`Text`** property of each **`TextBox`** to retrieve the values that the user entered. Line 216 then invokes our **`Maximum`** method to determine which value is the largest. Method **`Maximum`** provides the largest number as its return value, which line 216 stores in **`double`** variable **`maximum`**. Line 219 appends the **`maximum`** value to the **`maximumLabel`**'s **`Text`** property to display the result to the user.

Now let us examine the implementation of method **`Maximum`** (lines 195–198). The first line indicates that the method returns a **`double`** floating-point value, that the method's name is **`Maximum`** and that the method takes three **`double`** parameters (**`x`**, **`y`** and **`z`**). The statement in the body of the method (line 197) returns the largest of the three floating-point values using two calls to method **`Math.Max`**. First, method **`Math.Max`** is invoked and passed the values of variables **`y`** and **`z`** to determine the larger of these two values. Next, the value of variable **`x`** and the result of the first call to **`Math.Max`** are passed to method **`Math.Max`**. Finally, the result of the second call to **`Math.Max`** is returned to the caller.

6.6 Argument Promotion

Another important feature of method definitions is the *coercion of arguments* (i.e., forcing arguments to the appropriate type to pass to a method). This process commonly is referred to as *implicit conversion*, in that a copy of the variable's value is converted to a different type without an explicit cast. *Explicit conversion* occurs when an explicit cast specifies that conversion is to occur. Such conversions also can be done with class ***`Convert`*** in namespace **`System`**. C# supports both widening and narrowing conversions—*widening conversion* occurs when a type is converted to other types (usually types that can hold more data) without losing data, and a *narrowing conversion* occurs when data may be lost through a conversion (usually to types that hold a smaller amount of data). Figure 6.5 shows allowed implicit conversions.

Type	Can be Converted to Type(s)
`bool`	`object`
`byte`	`decimal`, `double`, `float`, `int`, `uint`, `long`, `ulong`, `object`, `short` or `ushort`
`sbyte`	`decimal`, `double`, `float`, `int`, `long`, `object` or `short`
`char`	`decimal`, `double`, `float`, `int`, `uint`, `long`, `ulong`, `object` or `ushort`
`decimal`	`object`
`double`	`object`
`float`	`double` or `object`
`int`	`decimal`, `double`, `float`, `long` or `object`
`uint`	`decimal`, `double`, `float`, `long`, `ulong`, or `object`
`long`	`decimal`, `double`, `float` or `object`
`ulong`	`decimal`, `double`, `float` or `object`
`short`	`decimal`, `double`, `float`, `int`, `long` or `object`
`ushort`	`decimal`, `double`, `float`, `int`, `uint`, `long`, `ulong` or `object`

Fig. 6.5 Allowed implicit conversions.

For example, the **`Math`** class method **`Sqrt`** can be called with an integer argument, even though the method is defined in class **`Math`** to receive a **`double`** argument. The statement

```
Console.WriteLine( Math.Sqrt( 4 ) );
```

correctly evaluates **`Math.Sqrt( 4 )`** and displays the value **`2`**. C# implicitly converts the **`int`** value 4 to the **`double`** value 4.0 before passing the value to **`Math.Sqrt`**. In many cases, C# applies implicit conversions to argument values that do not correspond precisely to the parameter types in the method definition. In some cases, attempting these conversions leads to compiler errors because C# uses conversion rules to determine when a widening conversion can occur. In our previous **`Math.Sqrt`** example, C# converts an **`int`** to a **`double`** without changing its value. However, converting a **`double`** to an **`int`** truncates the fractional part of the **`double`** value. Converting large integer types to small integer types (e.g., **`long`** to **`int`**) also can result in changed values. Such narrowing conversions can lose data; therefore, C# does not allow narrowing conversions without an explicit cast operation.

The conversion rules apply to expressions containing values of two or more data types (also referred to as *mixed-type expressions*) and to primitive data-type values passed as arguments to methods. C# converts the type of each value in a mixed-type expression to the “highest” type in the expression. C# creates a temporary copy of each value and uses it in the expression—the original values remain unchanged. A method argument’s type can be promoted to any “higher” type.

Converting values to lower types can result in data loss. In cases where information could be lost through conversion, the compiler requires the programmer to use a cast to force the conversion to occur. To invoke our **`Square`** method, which takes an integer parameter (Fig. 6.3) with the **`double`** variable **`y`**, the method call would be written as

```
int result = Square( ( int ) y );
```

This statement explicitly casts (converts) a copy of the value of **`y`** to an integer for use in method **`Square`**. Thus, if **`y`**’s value is **`4.5`**, method **`Square`** returns **`16`**, not **`20.25`**.

Common Programming Error 6.11

When performing a narrowing conversion (e.g., **`double`** *to* **`int`***), converting a primitive-data-type value to another primitive data type may change the value. Also, converting any integral value to a floating-point value and back to an integral value may introduce rounding errors into the result.*

6.7 C# Namespaces

As we have seen, C# contains many predefined classes that are grouped into namespaces. Collectively we refer to this preexisting code as the Framework Class Library. The actual code for the classes is located in **`.dll`** files called assemblies.

Throughout the text, **`using`** statements specify the namespaces we use in each program. For example, a program includes the statement

```
using System;
```

to tell the compiler that we are using the **`System`** namespace. This **`using`** statement allows us to write **`Console.WriteLine`** rather than **`System.Console.WriteLine`** throughout the program. To use a class in a particular namespace, we must add a reference to the appropriate assembly (demonstrated in Section 3.2). Assembly references for namespace **`System`** are added automatically—other assemblies must be added explicitly.

We exercise a large number of the FCL classes in this book. Figure 6.6 lists a subset of the many namespaces in the FCL and provides a brief description of each. We use classes from these namespaces and others throughout the book. This table introduces readers to the variety of reusable components in the FCL. When learning C#, spend time reading the descriptions of the classes in the documentation.

Namespace	**Description**
`System`	Contains essential classes and data types (such as **`int`**, **`double`**, **`char`**, etc.). Implicitly referenced by all C# programs.
`System.Data`	Contains classes that form ADO .NET, used for database access and manipulation.
`System.Drawing`	Contains classes used for drawing and graphics.
`System.IO`	Contains classes for the input and output of data, such as with files.
`System.Threading`	Contains classes for multithreading, used to run multiple parts of a program simultaneously.
`System.Windows.Forms`	Contains classes used to create graphical user interfaces.
`System.Xml`	Contains classes used to process XML data.

Fig. 6.6 Namespaces in the Framework Class Library.

The set of namespaces available in the FCL is quite large. In addition to the namespaces summarized in Fig. 6.6, the FCL includes namespaces for complex graphics, advanced graphical user interfaces, printing, advanced networking, security, multimedia, accessibility (for people with disabilities) and many more. For an overview of the namespaces in the FCL, look up ".NET Framework, class library" in the help documentation.

6.8 Value Types and Reference Types

In the next section, we will discuss passing arguments to methods by value and by reference. To understand this, we first need to make a distinction between types in C#. Data types are either *value types* or *reference types*. A variable of a value type contains data of that type. A variable of a reference type, in contrast, contains the address of the location in memory where the data are stored. Value types normally represent single pieces of data, such as **`int`** or **`bool`** values. Reference types, on the other hand, refer to objects, which can contain many individual pieces of data. We discuss objects in detail in Chapters 8, 9 and 10 (Object-Based Programming, and Object-Oriented Programming parts 1 and 2).

C# includes built-in value types and reference types. The built-in value types are the *integral types* (**`sbyte`**, **`byte`**, **`char`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`** and **`ulong`**), the *floating-point types* (**`float`** and **`double`**) and the types **`decimal`** and **`bool`**. The built-in reference types are **`string`** and **`object`**. Programmers also can create value types and reference types. The reference types that programmers can create are classes (Chapter 8), interfaces (Chapter 8) and delegates (Chapter 9).

The table in Fig. 6.7 lists the primitive data types, which are building blocks for more complicated types. Like its predecessor languages C and C++, C# requires all variables to have a type before they can be used in a program. For this reason, C# is referred to as a *strongly typed language*.

Type	Size in bits	Values	Standard
`bool`	8	**`true`** or **`false`**	
`char`	16	**`'\u0000'`** to **`'\uFFFF'`**	(Unicode character set)
`byte`	8	0 to 255	(unsigned)
`sbyte`	8	-128 to +127	
`short`	16	–32,768 to +32,767	
`ushort`	16	0 to 65,535	(unsigned)
`int`	32	–2,147,483,648 to 2,147,483,647	
`uint`	32	0 to 4,294,967,295	(unsigned)
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	
`ulong`	64	0 to 18,446,744,073,709,551,615	(unsigned)
`decimal`	128	$1.0 \text{ x } 10^{-28}$ to $7.9 \text{ x } 10^{28}$	
`float`	32	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	(IEEE 754 floating point)

Fig. 6.7 C# built-in data types. (Part 1 of 2.)

Type	Size in bits	Values	Standard
`double`	64	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	(IEEE 754 floating point)
`object`			
`string`			(Unicode character set)

Fig. 6.7 C# built-in data types. (Part 2 of 2.)

In C and C++ programs, programmers frequently must write separate program versions to support different computer platforms because the primitive data types are not guaranteed to be identical from computer to computer. For example, an `int` value on one computer might occupy 16 bits (2 bytes) of memory, whereas an `int` value on another computer might occupy 32 bits (4 bytes) of memory. In C#, `int` values are always 32 bits (4 bytes).

Portability Tip 6.1

Primitive data types in C# are portable across all platforms that support C#.

Each data type in the table is listed with its size in bits (there are 8 bits to a byte) and its range of values. The designers of C# wanted code to be portable; therefore, they chose to use internationally recognized standards for both character formats (Unicode) and floating-point numbers (IEEE 754).

6.9 Passing Arguments: Pass-by-Value vs. Pass-by-Reference

Two ways to pass arguments to methods in many programming languages are *pass-by-value* and *pass-by-reference*. When an argument is passed by value, the called method receives a *copy* of the argument's value.

Testing and Debugging Tip 6.2

With pass-by-value, changes to the called method's copy do not affect the original variable's value. This prevents some possible side effects that hinder the development of correct and reliable software systems.

When an argument is passed using pass-by-reference, the caller gives the method the ability to access and modify the caller's original data directly. Pass-by-reference can improve performance because it eliminates the overhead of copying large data items such as objects; however, pass-by-reference can weaken security because the called method can modify the caller's data.

Software Engineering Observation 6.9

When returning information from a method via a `return` *statement, value-type variables always are returned by value (i.e., a copy is returned), and reference-type variables are always returned by reference (i.e., a reference to the object is returned).*

To pass an object reference into a method, simply specify the reference name in the method call. Then, in the method body, reference the object using the parameter name. This refers to the original object in memory, which allows the called method to access the original object directly.

In Section 6.8, we discussed the difference between value types and reference types. At this point, the reader can understand one of the major differences between the two data types—value-type variables are passed to methods by value, whereas reference-type variables are passed to methods by reference. What if the programmer would like to pass a value type by reference? To do this, C# provides the ***`ref`*** and ***`out`*** *keywords*. The **`ref`** keyword specifies that a value-type argument should be passed by reference, which enables the called method to modify the original variable. This keyword is used for variables that already have been initialized. The **`out`** keyword specifies an output parameter, which is an argument to which the called method will assign a value. Normally, when a method receives an uninitialized value, the compiler generates an error. Preceding the parameter with keyword **`out`** specifies that the called method will initialize the variable and prevents the compiler from generating an error message for the uninitialized variable. Figure 6.8 demonstrates using the **`ref`** and **`out`** keywords to manipulate integer values.[2]

This program contains three methods to calculate the square of an integer. The first method, **`SquareRef`** (lines 26–29), multiplies its argument **`x`** by itself and assigns the new value to **`x`**. **`SquareRef`** receives its argument as a **`ref int`**, specifying that **`x`** is an integer that is passed by reference to the method. As a result, the assignment at line 28 modifies the original argument's value, rather than a copy of that value.

The second method, **`SquareOut`** (lines 33–37), does the same thing, but initializes **`x`** to **`6`** on line 35. **`SquareOut`** receives its argument as an **`out int`**, which indicates that **`x`** is an integer variable that the caller passes to method **`SquareOut`** by reference and that **`SquareOut`** can assign a new value to this variable. The final method, **`Square`** (lines 41–44), simply takes **`x`** as a value-type integer argument and squares its value.

```
 1  // Fig. 6.8: RefOutTest.cs
 2  // Demonstrating ref and out parameters.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.Data;
 9
10  public class RefOutTest : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.Button showOutputButton;
13     private System.Windows.Forms.Label outputLabel;
14
15     // Visual Studio .NET generated code
16
17     // main entry point for application
18     [STAThread]
19     static void Main()
20     {
21        Application.Run( new RefOutTest() );
22     }
```

Fig. 6.8 Demonstrating **`ref`** and **`out`** parameters. (Part 1 of 3.)

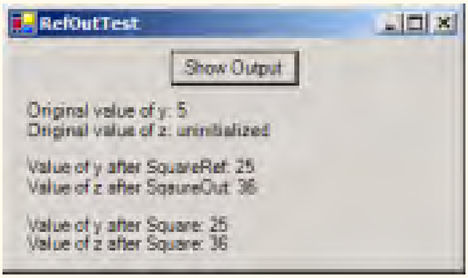
---

[2] In Chapter 7 we discuss passing reference-type arguments by value and by reference.

```

      // x passed by reference and method modifies
      // original variable's value
      void SquareRef( ref int x )
      {
         x = x * x;
      }

      // x passed as out parameter and method initializes
      // and modifies original variable's value
      void SquareOut( out int x )
      {
         x = 6;
         x = x * x;
      }

      // x passed by value and method cannot modify
      // original variable's value
      void Square( int x )
      {
         x = x * x;
      }

      private void showOutputButton_Click( object sender,
         System.EventArgs e )
      {
         int y = 5; // create new int and initialize to 5
         int z;     // declare z, but do not initialize it

         // display original values of y and z
         outputLabel.Text = "Original value of y: " + y + "\n";
         outputLabel.Text +=
            "Original value of z: uninitialized\n\n";

         // pass y and z by reference
         SquareRef( ref y );
         SquareOut( out z );

         // display values of y and z after modified by methods
         // SquareRef and SquareOut
         outputLabel.Text +=
            "Value of y after SquareRef: " + y + "\n";
         outputLabel.Text +=
            "Value of z after SqaureOut: " + z + "\n\n";

         // pass y and z by value
         Square( y );
         Square( z );

         // display unchanged values of y and z
         outputLabel.Text += "Value of y after Square: " + y + "\n";
         outputLabel.Text += "Value of z after Square: " + z + "\n";

```

Fig. 6.8 Demonstrating `ref` and `out` parameters. (Part 2 of 3.)

```
76
77      } // end method showOutputButton_Click
78   }
```



Original value of y: 5
Original value of z: uninitialized

Value of y after SquareRef: 25
Value of z after SquareOut: 36

Value of y after Square: 25
Value of z after Square: 36

Fig. 6.8 Demonstrating `ref` and `out` parameters. (Part 3 of 3.)

Method **`showOutputButton_Click`** (lines 46–77) is an event handler that invokes methods **`SquareRef`**, **`SquareOut`** and **`Square`** when the user clicks the **Show Output** button. This method begins by initializing **`y`** to **`5`** and declaring (but not initializing) **`z`**. Lines 58–59 call methods **`SquareRef`** and **`SquareOut`**. Notice the syntax used for passing **`y`** and **`z`**—in each case, we precede the argument either with **`ref`** or with **`out`**. The output displays the values of **`y`** and **`z`** after the function calls. Notice that **`y`** has been changed to **`25`** and **`z`** has been set to **`36`**. Finally, on lines 69–70 we call method **`Square`**. Arguments **`y`** and **`z`** both are passed by value—only copies of their values are passed to the method. As a result, the values of **`y`** and **`z`** remain **`25`** and **`36`**, respectively.

Common Programming Error 6.12

*The **`ref`** and **`out`** arguments in a method call must match those specified in the method definition; otherwise, a syntax error occurs.*

Software Engineering Observation 6.10

By default, C# does not allow the programmer to choose whether to pass each argument by value or by reference. Value-type variables are passed by value. Objects are not passed to methods; rather, references to objects are passed to methods. The references themselves are passed by value. When a method receives a reference to an object, the method can manipulate the object directly, but the reference value cannot be changed (e.g., to refer to a new object).

6.10 Random-Number Generation

We now take a brief and hopefully entertaining diversion into a popular programming application—simulation and game playing. In this section and the next, we develop a nicely structured game-playing program that includes multiple methods. The program uses most of the control structures we have studied to this point and also introduces several new concepts.

There is something in the air of a gambling casino that invigorates every type of person—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It is the *element of chance*, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced into computer applications with the ***`Random`*** class (located in namespace **`System`**).

Consider the following statements:

```
Random randomObject = new Random();
int randomNumber = randomObject.Next();
```

The **Next** method generates a positive **int** value between zero and the constant **Int32.MaxValue** (the value 2,147,483,647). If **Next** produces values at random, every value in this range has an equal *chance* (or *probability*) of being chosen when **Next** is called. Note that values returned by **Next** are actually *pseudo-random numbers*—a sequence of values produced by a complex mathematical calculation. A *seed* value is required in this mathematical calculation. When we create our **Random** object, we use the current time of day as the seed. A particular seed value always produces the same series of random numbers. Programmers commonly use the current time of day as a seed value, since it changes each second and, therefore produces different random-number sequences each time the program executes.

The range of values produced directly by **Next** often is different from the range of values required in a particular application. For example, a program that simulates coin-tossing might require only 0 for “heads” and 1 for “tails.” A program that simulates rolling a six-sided die would require random integers in the range 1–6. A video-game program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon might require random integers in the range 1–4.

The one-argument version of method **Next** returns values in the range from 0 up to (but not including) the value of that argument. For example,

```
value = randomObject.Next( 6 );
```

produces values from 0 through 5. This is called *scaling*, because the range of values produced has been scaled down from over two billion to only six. The number 6 is the *scaling factor*. The two-argument version of method Next allows us to *shift* and scale the range of numbers. For example, we can use method **Next** as follows

```
value = randomObject.Next( 1, 7 );
```

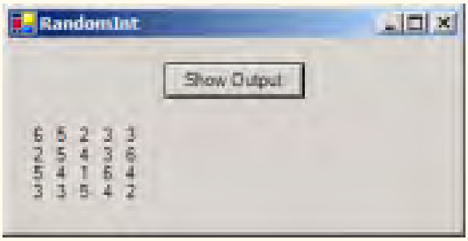
to produce integers in the range from 1 to 6. In this case, we have shifted the numbers to produce a range from 1 up to (but not including) 7.

The Windows application of Fig. 6.9 simulates 20 rolls of a six-sided die and shows the integer value of each roll. The dice-rolling simulation begins when the user clicks the **Show Ouput** button, which invokes the **showOutputButton_Click** event handler (lines 24–44). The **for** loop on lines 32–43 repeatedly invokes method **Next** of class **Random** to simulate rolling the die. Lines 37–38 append the value rolled to **outputLabel**’s **Text** property. After every five rolls, line 42 appends a newline character to make the output more readable.

```
1  // Fig. 6.9: RandomInt.cs
2  // Generating random integer values.
3  using System;
4  using System.Drawing;
```

Fig. 6.9 Random integers in the range 1–6. (Part 1 of 2.)

```
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class RandomInt : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button showOutputButton;
   private System.Windows.Forms.Label outputLabel;

   // Visual Studio .NET generated code

   // the main entry point for the application
   [STAThread]
   static void Main()
   {
      Application.Run( new RandomInt() );
   }

   private void showOutputButton_Click( object sender,
      System.EventArgs e )
   {
      Random randomInteger = new Random();

      outputLabel.Text = "";

      // loop 20 times
      for ( int counter = 1; counter <= 20; counter++ )
      {
         // pick random integer between 1 and 6
         int nextValue = randomInteger.Next( 1, 7 );

         outputLabel.Text +=
            nextValue + "   "; // append value to output

         // add newline after every 5 values
         if ( counter % 5 == 0 )
            outputLabel.Text += "\n";
      }
   }
}
```



```
6  5  2  3  3
2  5  4  3  6
5  4  1  6  4
3  3  5  4  2
```

Fig. 6.9 Random integers in the range 1–6. (Part 2 of 2.)

The Windows application of Fig. 6.10 simulates rolls of four dice. The program enables the user to click a button that “rolls” four dice at a time and displays an image of each die in the window. The next example (Fig. 6.11) uses many of this example’s features to demonstrate that the numbers generated by `Next` occur with approximately equal likelihood.

```
// Fig. 6.10: RollDie.cs
// Using random number generation to simulate dice rolling.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;   // enables reading data from files

public class RollDie : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button rollButton;

   private System.Windows.Forms.Label dieLabel2;
   private System.Windows.Forms.Label dieLabel1;
   private System.Windows.Forms.Label dieLabel3;
   private System.Windows.Forms.Label dieLabel4;

   private Random randomNumber = new Random();

   // Visual Studio .NET generated code

   // method called when rollButton clicked,
   // passes labels to another method
   protected void rollButton_Click(
      object sender, System.EventArgs e )
   {
      // pass the labels to a method that will
      // randomly assign a face to each die
      DisplayDie( dieLabel1 );
      DisplayDie( dieLabel2 );
      DisplayDie( dieLabel3 );
      DisplayDie( dieLabel4 );

   } // end rollButton_Click

   // determines image to be displayed by current die
   public void DisplayDie( Label dieLabel )
   {
      int face = randomNumber.Next( 1, 7 );

      // displays image specified by filename
      dieLabel.Image = Image.FromFile(
         Directory.GetCurrentDirectory() +
         "\\images\\die" + face + ".gif" );
   }
```

Fig. 6.10 Rolling dice in a Windows application (Part 1 of 2.).

```
48
49    // main entry point for application
50    [STAThread]
51    static void Main()
52    {
53       Application.Run( new RollDie() );
54    }
55
56 } // end class RollDie
```

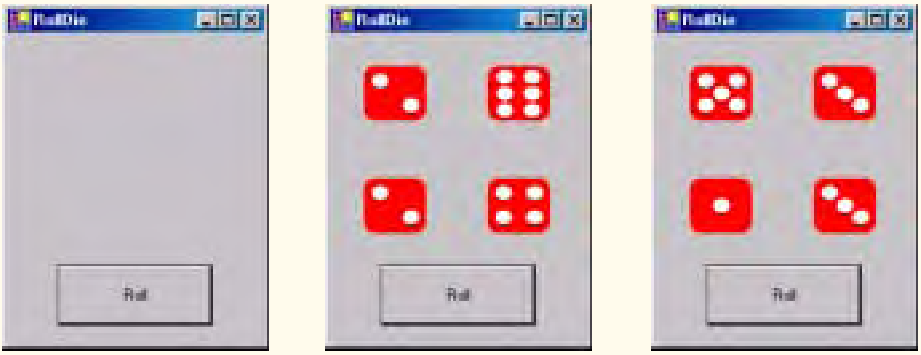


Fig. 6.10 Rolling dice in a Windows application (Part 2 of 2.).

Method **`DisplayDie`** (lines 39–47) invokes **`Random`** method **`Next`** to simulate a roll of a die (line 41) and loads an image that corresponds to the value rolled (lines 44–46). Line 44 uses class **`Label`**'s **`Image`** property (introduced in Chapter 2) to display the die. Notice that we specify which image will be displayed by invoking method ***`FromFile`*** of class **`Image`**, which specifies the location of the file on disk that contains the image. Each click of the button displays four images that represent the four new values of the dice. Note that the user must click **`rollButton`** at least once to display the dice. **`Directory`** method **`GetCurrentDirectory`** (line 45) returns the path of the folder in which the program is executing. If you run the program from Visual Studio .NET, this will be the **`bin\Debug`** directory in the project's directory. The die images must be in this folder for the example to operate properly. These images are placed in the proper folders on the CD that accompanies this book.

To show that class **`Random`** produces numbers with approximately equal likelihood, let us modify the program in Fig. 6.10 to keep some simple statistics. The Windows application of Fig. 6.11 provides a **Roll** button for rolling the dice and a **`TextBox`** that displays the frequencies for each value rolled. The program output shows the results of clicking **Roll** 10 times.

When the user clicks the **Roll** button, the program invokes the **`rollButton_Click`** event handler on lines 38–73. This event handler invokes method **`DisplayDie`** for each of the 12 dice that the program simulates (lines 43–54). Lines 56–71 then calculate the frequencies for each die and displays the results by appending the information to **`displayTextBox`**'s **`Text`** property. Method **`displayDie`** (lines 76–113) simulates a die roll (line 78), loads the appropriate **`Image`** and increments the frequency count for the rolled value.

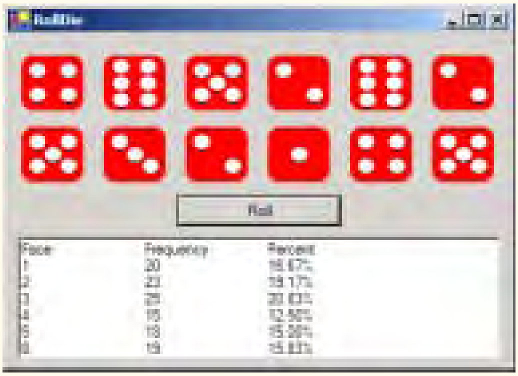
```
// Fig. 6.11: RollDie2.cs
// Rolling 12 dice with frequency chart.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

public class RollDie2 : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button rollButton;

   private System.Windows.Forms.RichTextBox displayTextBox;

   private System.Windows.Forms.Label dieLabel1;
   private System.Windows.Forms.Label dieLabel2;
   private System.Windows.Forms.Label dieLabel3;
   private System.Windows.Forms.Label dieLabel4;
   private System.Windows.Forms.Label dieLabel5;
   private System.Windows.Forms.Label dieLabel6;
   private System.Windows.Forms.Label dieLabel7;
   private System.Windows.Forms.Label dieLabel8;
   private System.Windows.Forms.Label dieLabel9;
   private System.Windows.Forms.Label dieLabel10;
   private System.Windows.Forms.Label dieLabel11;
   private System.Windows.Forms.Label dieLabel12;

   private Random randomNumber = new Random();

   private int ones, twos, threes, fours, fives, sixes;

   // Visual Studio .NET generated code

   // simulates roll by calling DisplayDie for
   // each label and displaying the results
   protected void rollButton_Click(
      object sender, System.EventArgs e )
   {
      // pass the labels to a method that will
      // randomly assign a face to each die
      DisplayDie( dieLabel1 );
      DisplayDie( dieLabel2 );
      DisplayDie( dieLabel3 );
      DisplayDie( dieLabel4 );
      DisplayDie( dieLabel5 );
      DisplayDie( dieLabel6 );
      DisplayDie( dieLabel7 );
      DisplayDie( dieLabel8 );
      DisplayDie( dieLabel9 );
      DisplayDie( dieLabel10 );
```

Fig. 6.11 Simulating rolling 12 six-sided dice. (Part 1 of 3.)

```
      DisplayDie( dieLabel11 );
      DisplayDie( dieLabel12 );

      double total = ones + twos + threes + fours + fives + sixes;

      // display the current frequency values
      displayTextBox.Text = "Face\t\tFrequency\tPercent\n1\t\t" +
         ones + "\t\t" +
         String.Format( "{0:F2}", ones / total * 100 ) +
         "%\n2\t\t" + twos + "\t\t" +
         String.Format( "{0:F2}", twos / total * 100 ) +
         "%\n3\t\t" + threes + "\t\t" +
         String.Format( "{0:F2}", threes / total * 100 ) +
         "%\n4\t\t" + fours + "\t\t" +
         String.Format( "{0:F2}", fours / total * 100 ) +
         "%\n5\t\t" + fives + "\t\t" +
         String.Format( "{0:F2}", fives / total * 100 ) +
         "%\n6\t\t" + sixes + "\t\t" +
         String.Format( "{0:F2}", sixes / total * 100 ) + "%";

   } // end rollButton_Click

   // display the current die, and modify frequency values
   public void DisplayDie( Label dieLabel )
   {
      int face = randomNumber.Next( 1, 7 );

      dieLabel.Image = Image.FromFile(
         Directory.GetCurrentDirectory() +
         "\\images\\die" + face + ".gif" );

      // add one to frequency of current face
      switch ( face )
      {
         case 1:
            ones++;
            break;

         case 2:
            twos++;
            break;

         case 3:
            threes++;
            break;

         case 4:
            fours++;
            break;

         case 5:
            fives++;
            break;
```

Fig. 6.11 Simulating rolling 12 six-sided dice. (Part 2 of 3.)

```
         case 6:
            sixes++;
            break;

      } // end switch

   } // end DisplayDie

   // main entry point for the application
   [STAThread]
   static void Main()
   {
      Application.Run( new RollDie2() );
   }

} // end of class RollDie2
```



```
Face        Frequency        Percent
1           20               16.67%
2           23               19.17%
3           25               20.83%
4           15               12.50%
5           18               15.00%
6           19               15.83%
```

Fig. 6.11 Simulating rolling 12 six-sided dice. (Part 3 of 3.)

As the program output demonstrates, over a large number of die rolls, each of the possible faces from 1 through 6 appears with approximately equal likelihood (i.e., about one-sixth of the time). After studying arrays in Chapter 7, Arrays, we will show how to replace the entire `switch` structure in this program with a single-line statement.

6.11 Example: Game of Chance

One of the most popular games of chance is a dice game known as “craps,” played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. Each face contains 1, 2, 3, 4, 5 or 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes the player’s “point.” To win, players must continue rolling the dice until they “make their point” (i.e., roll their point value). The player loses by rolling a 7 before making the point.*

Figure 6.12 simulates the game of craps with a simple graphical user interface.

Notice that the player rolls two dice on each roll. When executing the application, clicking the **Play** button begins the game and makes the first roll. The form displays the results of each roll. The screen captures show the execution of several games.

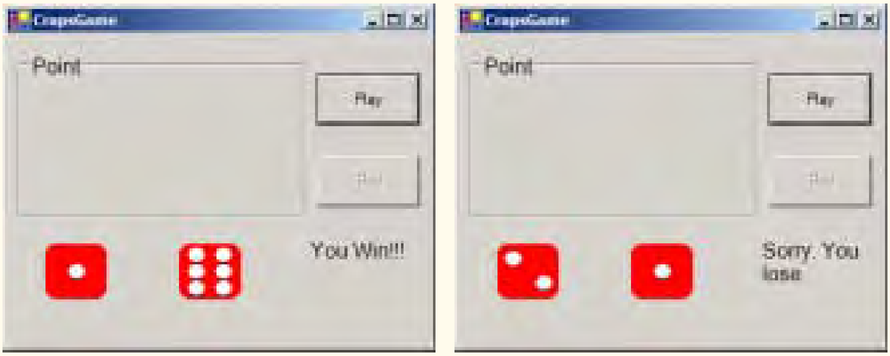
```
// Fig. 6.12: CrapsGame.cs
// Simulating the game of Craps.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

public class CrapsGame : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button rollButton;
   private System.Windows.Forms.Button playButton;

   int myPoint; // player's point value
   private System.Windows.Forms.PictureBox pointFirstDieImage;
   private System.Windows.Forms.Label statusLabel;
   private System.Windows.Forms.PictureBox firstDieImage;
   private System.Windows.Forms.PictureBox pointSecondDieImage;
   private System.Windows.Forms.PictureBox secondDieImage;
   private System.Windows.Forms.GroupBox pointGroupBox;
   int myDie1; // value of first die
   int myDie2; // value of second die

   public enum DiceNames
   {
      SNAKE_EYES = 2,
      TREY = 3,
      YO_LEVEN = 11,
      BOX_CARS = 12,
   }

   // Visual Studio .NET generated code

   // simulate next roll and result of that roll
   protected void rollButton_Click(
      object sender, System.EventArgs e )
   {
      int sum = rollDice();

      if ( sum == myPoint )
      {
         statusLabel.Text = "You Win!!!";
         rollButton.Enabled = false;
         playButton.Enabled = true;
      }
```

Fig. 6.12 Program to simulate the game of craps. (Part 1 of 4.)

```
         else
            if ( sum == 7 )
            {
               statusLabel.Text = "Sorry. You lose.";
               rollButton.Enabled = false;
               playButton.Enabled = true;
            }

      } // end rollButton_Click

      // simulate first roll and result of that roll
      protected void playButton_Click(
         object sender, System.EventArgs e )
      {
         pointGroupBox.Text = "Point";
         statusLabel.Text = "";
         pointFirstDieImage.Image = null;
         pointSecondDieImage.Image = null;

         myPoint = 0;
         int sum = rollDice();

         switch ( sum )
         {
            case 7:
            case ( int ) DiceNames.YO_LEVEN:
               rollButton.Enabled = false;  // disable Roll button
               statusLabel.Text = "You Win!!!";
               break;

            case ( int ) DiceNames.SNAKE_EYES:
            case ( int ) DiceNames.TREY:
            case ( int ) DiceNames.BOX_CARS:
               rollButton.Enabled = false;
               statusLabel.Text = "Sorry. You lose.";
               break;

            default:
               myPoint = sum;
               pointGroupBox.Text = "Point is " + sum;
               statusLabel.Text = "Roll Again";
               displayDie( pointFirstDieImage, myDie1 );
               displayDie( pointSecondDieImage, myDie2 );
               playButton.Enabled = false;
               rollButton.Enabled = true;
               break;

         } // end switch

      } // end playButton_Click

```

Fig. 6.12 Program to simulate the game of craps. (Part 2 of 4.)

```
   // display an image for the specified face
   private void displayDie( PictureBox dieImage, int face )
   {
      dieImage.Image = Image.FromFile(
         Directory.GetCurrentDirectory() +
         "\\images\\die" + face + ".gif" );
   }

   // simulates rolling two dice
   private int rollDice()
   {
      int die1, die2, dieSum;
      Random randomNumber = new Random();

      die1 = randomNumber.Next( 1, 7 );
      die2 = randomNumber.Next( 1, 7 );

      displayDie( firstDieImage, die1 );
      displayDie( secondDieImage, die2 );

      myDie1 = die1;
      myDie2 = die2;
      dieSum = die1 + die2;
      return dieSum;

   } // end method rollDice

   // main entry point for the application
   [STAThread]
   static void Main()
   {
      Application.Run( new CrapsGame() );
   }

} // end of class CrapsGame
```



You Win!!!

Sorry. You lose

Fig. 6.12 Program to simulate the game of craps. (Part 3 of 4.)

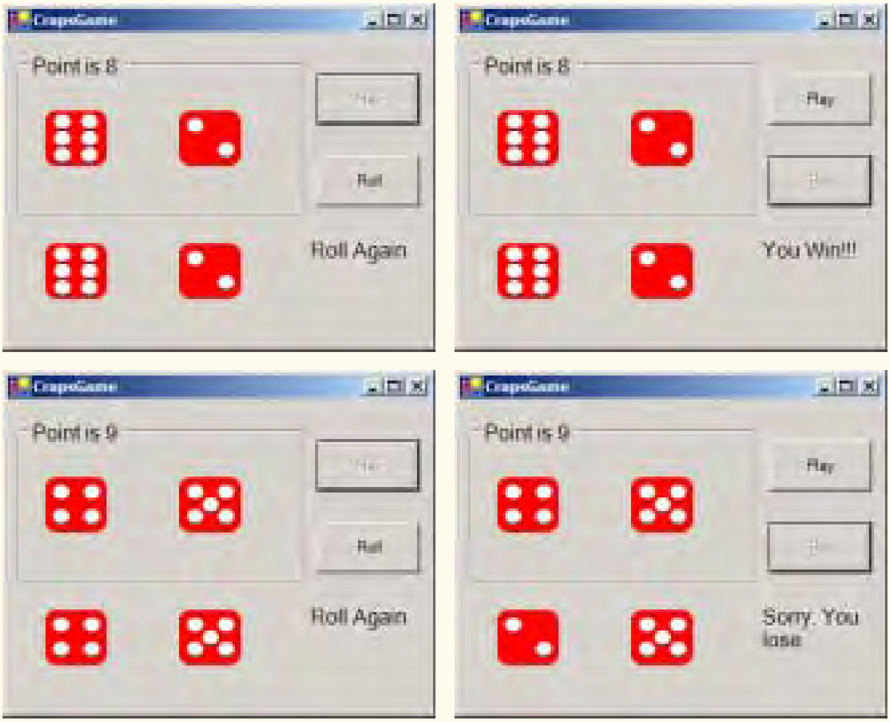


Fig. 6.12 Program to simulate the game of craps. (Part 4 of 4.)

Before its method definitions, the program includes several declarations, including an *enumeration* on lines 26–32. An enumeration is a value type that contains a set of constant values and is created using the keyword **`enum`**. This enumeration is a convenient way of referring to constant values used throughout the program. We have used the identifiers **`SNAKE_EYES`**, **`TREY`**, **`CRAPS`**, **`YO_LEVEN`** and **`BOX_CARS`**, to represent significant values in craps. Using these identifiers makes the program more readable. Additionally, if we need to change one of these values, we can modify the enumeration instead of changing the values where they are used throughout the program.

This example introduces a few new GUI components. The first, called a ***`GroupBox`***, displays the user's point. A **`GroupBox`** is a container for other components and helps group components logically. Within the **`GroupBox`**, we add two **`PictureBox`**es—components that display images. These are added, as with other components, by clicking **`PictureBox`** in the **`ToolBox`** and dragging this component within the borders of the **`GroupBox`**.

The **`playButton_Click`** event handler begins the game. Line 68 invokes method **`rollDice`** (defined on lines 108–124), which rolls the dice, displays the dice and returns their sum. Lines 70–95 use a **`switch`** structure to determine whether the player won, lost or established a point value. If the player won by rolling a 7 or 11, line 74 disables **`rollButton`** to prevent the player from rolling the dice again. Line 75 displays a message to indicate that the user won. If the player lost by rolling **`SNAKE_EYES`**, **`TREY`** or **`BOX_CARS`** (i.e., 2, 3 or 12), line 82 displays a message to indicate that the user lost. Otherwise, the default case (lines 85–93) sets the player's point, displays the dice in `pointGroupBox`, enables `rollButton` and disables `playButton`. Notice that for many of the `case`s, we cast the enumeration values to type `int`. Although each enumeration value is assigned an integer value on lines 26–32, each value is considered to be of `enum` type `DiceNames`, and therefore must be cast to `int` for use in the `switch` structure, which requires constant integral expressions.

The `rollButton_Click` event handler's task is to roll the dice and determine if the player won by making the point value or lost by rolling 7. Line 40 calls method `rollDice`. Lines 42–54 in method `rollButton_Click` analyze the roll. Depending on the value of the roll, the buttons `rollButton` and `playButton` will become either disabled or enabled. This is done by setting the `Enabled` property to `true` or `false`.

6.12 Duration of Variables

The attributes of variables include name, type, size and value. Each variable in a program has additional attributes, including *duration* and *scope*.

A variable's duration (also called its *lifetime*) is the period during which the variable exists in memory. Some variables exist briefly, some are created and destroyed repeatedly and others exist for the entire execution of a program.

A variable's *scope* is where the variable's identifier (i.e., name) can be referenced in a program. Some variables can be referenced throughout a program, while others can be referenced from limited portions of a program. This section discusses the duration of variables. Section 6.13 discusses the scope of identifiers.

Local variables in a method (i.e., parameters and variables declared in the method body) have *automatic duration*. Automatic duration variables are created when program control reaches their declaration; that is, they exist while the block in which they are declared is active, and they are destroyed when that block is exited. For the remainder of the text, we refer to variables of automatic duration as automatic variables, or local variables.

The instance variables of a class are initialized by the compiler if the programmer does not provide initial values. Variables of most primitive data types are initialized to zero, `bool` variables are initialized to `false` and references are initialized to `null`. Unlike instance variables of a class, automatic variables must be initialized by the programmer before they can be used.

Common Programming Error 6.13

Automatic variables must be initialized before their values are used in a method; otherwise, the compiler issues an error message.

Variables of *static duration* exist from the time at which the class that defines them is loaded into memory. These variables then last until the program terminates. Their storage is allocated and initialized when their classes are loaded into memory. Static-duration variable names exist when their classes are loaded into memory, but this does not mean that these identifiers necessarily can be used throughout the program—their scopes may be limited as we will see in the next section.

6.13 Scope Rules

The *scope* (sometimes called *declaration space*) of an identifier for a variable, reference or method is the portion of the program in which the identifier can be accessed. A local variable or reference declared in a block can be used only in that block or in blocks nested within that block. The possible scopes for an identifier are *class scope* and *block scope*.

Members of a class have class scope and are visible in what is known as the *declaration space of a class*. Class scope begins at the opening left brace (**{**) of the class definition and terminates at the closing right brace (**}**). Class scope enables methods of a class to access all members defined in that class. In Chapter 8, Object-Based Programming, we see that **static** members are an exception to this rule. In a sense, all instance variables and methods of a class are *global* to the methods of the class in which they are defined (i.e., the methods can modify the instance variables directly and invoke other methods of the class).

Identifiers declared inside a block have block scope (*local-variable declaration space*). Block scope begins at the identifier's declaration and ends at the block's terminating right brace (**}**). Local variables of a method have block scope, as do method parameters, which are local variables of the method. Any block may contain variable declarations. When blocks are nested in a method's body, and an identifier declared in an outer block has the same name as an identifier declared in an inner block, an error is generated. On the other hand, if a local variable in a method has the same name as an instance variable, the value in the calling method (main program) is "hidden" until the method terminates execution. In Chapter 8, Object-Based Programming, we discuss how to access such "hidden" instance variables. The reader should note that block scope also applies to methods and **for** structures. With **for** structures, any variable declared in the initialization portion of the **for** header will be in scope only within that **for** structure.

Good Programming Practice 6.4

Avoid local-variable names that hide instance-variable names.

The program in Fig. 6.13 demonstrates scoping issues with instance variables and local variables. Instance variable **x** (line 15) is initialized to 1. This instance variable is hidden in any block (or method) that declares a local variable named **x**. The **showOutputButton_Click** event handler (lines 47–63) declares a local variable **x** and initializes it to 5 (line 50). Lines 52–53 display the value of this local variable to show that instance variable **x** (with value 1) is "hidden" in method **showOutputButton_Click**.

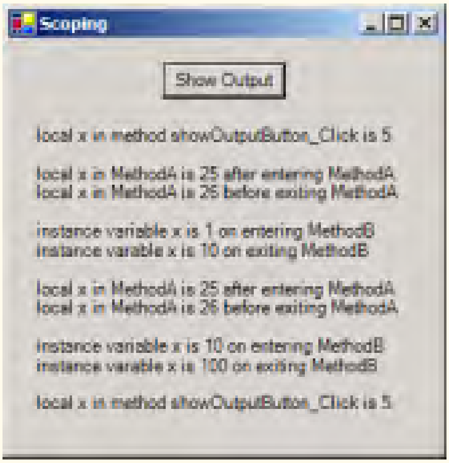
```
 1  // Fig. 6.13: Scoping.cs
 2  // Demonstrating scope of local and instance variables.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.Data;
 9
10  public class Scoping : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.Label outputLabel;
13     private System.Windows.Forms.Button showOutputButton;
14
15     public int x = 1; // instance variable
```

Fig. 6.13 Scoping. (Part 1 of 3.)

```

      // Visual Studio .NET generated code

      public void MethodA()
      {
         int x = 25;    // initialized each time a is called

         outputLabel.Text = outputLabel.Text +
            "\n\nlocal x in MethodA is " + x +
            " after entering MethodA";

         ++x; // increment local variable x

         outputLabel.Text = outputLabel.Text +
            "\nlocal x in MethodA is " + x +
            " before exiting MethodA";
      }

      public void MethodB()
      {
         outputLabel.Text = outputLabel.Text +
            "\n\ninstance variable x is " + x +
            " on entering MethodB";

         x *= 10;

         outputLabel.Text = outputLabel.Text +
            "\ninstance varable x is " + x +
            " on exiting MethodB";
      }

      private void showOutputButton_Click( object sender,
         System.EventArgs e )
      {
         int x = 5; // local x in method showOutputButton_Click

         outputLabel.Text =
            "local x in method showOutputButton_Click is " + x;

         MethodA();    // MethodA has automatic local x;
         MethodB();    // MethodB uses instance variable x
         MethodA();    // MethodA creates new automatic local x
         MethodB();    // instance variable x retains its value

         outputLabel.Text = outputLabel.Text + "\n\n" +
            "local x in method showOutputButton_Click is " + x;

      }  // end method showOutputButton_Click

      // main entry point for the application
      [STAThread]
      static void Main()
      {
```

Fig. 6.13 Scoping. (Part 2 of 3.)

```
69          Application.Run( new Scoping() );
70       }
71 
72    } // end of class Scoping
```



```
local x in method showOutputButton_Click is 5

local x in MethodA is 25 after entering MethodA
local x in MethodA is 26 before exiting MethodA

instance variable x is 1 on entering MethodB
instance variable x is 10 on exiting MethodB

local x in MethodA is 25 after entering MethodA
local x in MethodA is 26 before exiting MethodA

instance variable x is 10 on entering MethodB
instance variable x is 100 on exiting MethodB

local x in method showOutputButton_Click is 5
```

Fig. 6.13 Scoping. (Part 3 of 3.)

The program defines two other methods—**`MethodA`** and **`MethodB`**—that take no arguments and return nothing. The program calls each method twice from method **`Scoping`**. **`MethodA`** defines local variable **`x`** (line 21) and initializes it to **`25`**. Each call to **`MethodA`** displays the variable's value in **`outputLabel`**, increments the variable and displays it again before exiting the method. Each call to **`MethodA`** recreates automatic variable **`x`** and initializes it to **`25`**. Method **`MethodB`** does not declare any variables. Therefore, when it refers to variable **`x`**, the instance variable **`x`** is used. Each call to **`MethodB`** displays the instance variable in **`outputLabel`**, multiplies it by **`10`** (line 40) and displays it again before exiting the method. The next time method **`MethodB`** is called, the instance variable begins with its modified value, **`10`**. After the calls to **`MethodA`** and **`MethodB`**, the program again displays the local variable **`x`** in method **`showOutputButton_Click`** to show that none of the method calls modified this specific variable **`x`**, as the methods all referred to variables in other scopes.

6.14 Recursion

The programs we have discussed generally are structured as methods that call one another in a hierarchical manner. For some problems, it is useful to have a method actually call itself. A *recursive method* is a method that calls itself either directly or indirectly through another method. Recursion is an important topic discussed at length in upper-level computer science courses. In this section and the next, we present two simple examples of recursion. We consider recursion conceptually first, then examine several programs containing recursive methods.

Recursive problem-solving approaches have a number of elements in common. A recursive method is called to solve a problem. The method actually knows how to solve only the simplest case(s), or *base case(s)*. If the method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method divides the problem into two conceptual pieces—a piece that the method knows how to perform (base case) and a piece that the method does not know how to perform. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version of it. The method invokes (calls) a fresh copy of itself to work on the smaller problem—this is referred to as a *recursive call*, or a *recursion step*. The recursion step also normally includes the keyword **`return`**, because its result will be combined with the portion of the problem that the method knew how to solve. Such a combination will form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still "open" (i.e., it has not finished executing). The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces. Each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case, so the recursion can eventually terminate. At that point, the method recognizes the base case and returns a result to the previous copy of the method. A sequence of returns ensues up the line until the original method call returns the final result to the caller. As an example of these concepts, let us write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer *n*, written *n!* (and pronounced "*n* factorial"), is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

with 1! equal to 1, and 0! defined as 1. For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120.

The factorial of an integer **`number`** greater than or equal to 0 can be calculated *iteratively* (nonrecursively) using **`for`** as follows:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

We arrive at a recursive definition of the factorial method with the following relationship:

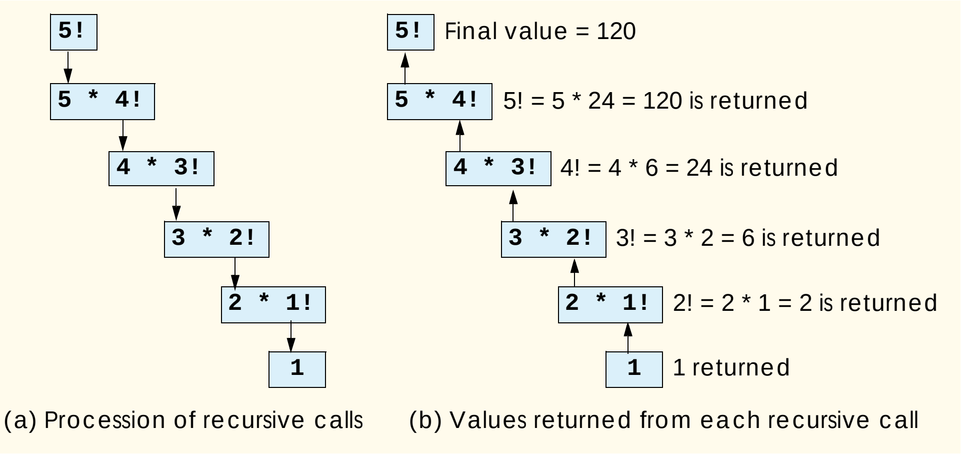
$$n! = n \cdot (n - 1)!$$

For example, 5! is clearly equal to 5 * 4!, as shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

A recursive evaluation of 5! would proceed as in Fig. 6.14. Figure 6.14a shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Each rectangle represents a method call. Figure 6.14 shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 6.15 uses recursion to calculate and print the factorials of the integers 0–10. The recursive method **`Factorial`** (lines 17–24) first determines whether its terminating condition is **`true`** (i.e., **`number`** is less than or equal to 1). If **`number`** is less than or equal to 1, **`factorial`** returns 1, no further recursion is necessary and the method


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
5!
Final value = 120
5 * 4!
5! = 5 * 24 = 120 is returned
4 * 3!
4! = 4 * 6 = 24 is returned
3 * 2!
3! = 3 * 2 = 6 is returned
2 * 1!
2! = 2 * 1 = 2 is returned
1
1 returned

(a) Procession of recursive calls (b) Values returned from each recursive call

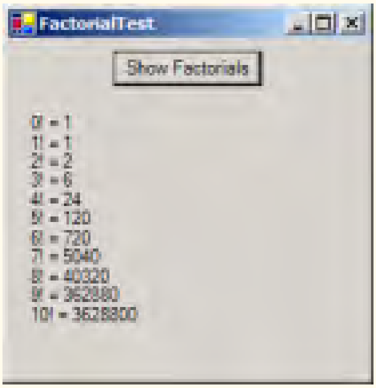
Fig. 6.14 Recursive evaluation of 5!.

returns. If **`number`** is greater than 1, line 23 expresses the problem as the product of **`number`** and a recursive call to **`Factorial`**, evaluating the factorial of **`number - 1`**. Note that **`Factorial( number - 1 )`** is a slightly simpler problem than the original calculation **`Factorial( number )`**.

```
1  // Fig. 6.15: FactorialTest.cs
2  // Calculating factorials with recursion.
3  using System;
4  using System.Drawing;
5  using System.Collections;
6  using System.ComponentModel;
7  using System.Windows.Forms;
8  using System.Data;
9
10 public class FactorialTest : System.Windows.Forms.Form
11 {
12    private System.Windows.Forms.Button showFactorialsButton;
13    private System.Windows.Forms.Label outputLabel;
14
15    // Visual Studio .NET generated code
16
17    public long Factorial( long number )
18    {
19       if ( number <= 1 )      // base case
20          return 1;
21
22       else
23          return number * Factorial( number - 1 );
24    }
25
```

Fig. 6.15 Calculating factorials with a recursive method. (Part 1 of 2.)

```
   // main entry point for the application
   [STAThread]
   static void Main()
   {
      Application.Run( new FactorialTest());
   }

   private void showFactorialsButton_Click( object sender,
      System.EventArgs e )
   {
      outputLabel.Text = "";

      for ( long i = 0; i <= 10; i++ )
         outputLabel.Text += i + "! = " +
            Factorial( i ) + "\n";
   }

} // end of class FactorialTest
```



```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 6.15 Calculating factorials with a recursive method. (Part 2 of 2.)

Method **`Factorial`** receives a parameter of type **`long`** and returns a result of type **`long`**. As seen in Fig. 6.15, factorial values become large quickly. We choose data type **`long`** so the program can calculate factorials greater than 20!. Unfortunately, the **`Factorial`** method produces large values so quickly, even **`long`** does not help us print many more factorial values before the size of even the **`long`** variable is exceeded.

Factorials of larger numbers require the program to use **`float`** and **`double`** variables. This points to a weakness in most programming languages, namely, that the languages are not easily extended to handle the unique requirements of various applications. As we will see in our treatment of object-oriented programming beginning in Chapter 8, C# is an extensible language—programmers with unique requirements can extend the language with new data types (called classes). A programmer could create a **`HugeInteger`** class, for example, that would enable a program to calculate the factorials of arbitrarily large numbers.

Common Programming Error 6.14

Forgetting to return a value from a recursive method can result in syntax and/or logic errors.

Common Programming Error 6.15

Omitting the base case or writing the recursion step so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. Infinite recursion is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

6.15 Example Using Recursion: The Fibonacci Series

The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…. This number, too, repeatedly occurs in nature and has been called the *golden ratio* or the *golden mean*. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards often are designed with a golden mean width-to-height ratio.

The recursive definition of the Fibonacci series is as follows:

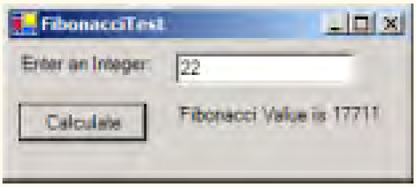
$$Fibonacci(\ 0\ ) = 0$$
$$Fibonacci(\ 1\ ) = 1$$
$$Fibonacci(\ n\ ) = Fibonacci(\ n - 1\ ) + Fibonacci(\ n - 2\ )$$

Note that there are two base cases for the Fibonacci calculation—*fibonacci(0)* evaluates to 0, and *fibonacci(1)* evaluates to 1. The application in Fig. 6.16 calculates the i^{th} Fibonacci number recursively using method **`Fibonacci`**. The user enters an integer in the text box, indicating the i^{th} Fibonacci number to calculate, and clicks the **`calculateButton`** (which displays the text **Calculate Fibonacci**). Method **`calculateButton_Click`** (lines 22–29) executes in response to the user interface event and calls recursive method **`Fibonacci`** to calculate the specified Fibonacci number. In Fig. 6.16, the screen captures show the results of calculating several Fibonacci numbers.

```
 1  // Fig. 6.16: FibonacciTest.cs
 2  // Recursive fibonacci method.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.Data;
 9
10  public class FibonacciTest : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.Button calculateButton;
13
14     private System.Windows.Forms.TextBox inputTextBox;
```

Fig. 6.16 Recursively generating Fibonacci numbers. (Part 1 of 2.)

```
15
16     private System.Windows.Forms.Label displayLabel;
17     private System.Windows.Forms.Label promptLabel;
18
19     // Visual Studio .NET generated code
20
21     // call Fibonacci and display results
22     protected void calculateButton_Click(
23        object sender, System.EventArgs e )
24     {
25        int number = Convert.ToInt32( inputTextBox.Text );
26        int fibonacciNumber = Fibonacci( number );
27        displayLabel.Text =
28           "Fibonacci Value is " + fibonacciNumber;
29     }
30
31     // calculates Fibonacci number
32     public int Fibonacci( int number )
33     {
34        if ( number == 0 || number == 1 )
35           return number;
36        else
37           return Fibonacci( number - 1 ) + Fibonacci( number - 2 );
38     }
39
40     // main entry point for the application
41     [STAThread]
42     static void Main()
43     {
44        Application.Run( new FibonacciTest() );
45     }
46
47  } // end of class FibonacciTest
```



Enter an Integer: 22

Fibonacci Value is 17711

Fig. 6.16 Recursively generating Fibonacci numbers. (Part 2 of 2.)

The call to **`Fibonacci`** (line 26) from **`calculateButton_Click`** is not a recursive call, but all subsequent calls to **`Fibonacci`** from line 37 are recursive. Each time **`Fibonacci`** is invoked, it immediately tests for the base case—**`number`** equal to 0 or 1 (line 34). If this is true, **`Fibonacci`** returns **`number`** (*fibonacci(0)* is 0 and *fibonacci(1)* is 1). Interestingly, if **`number`** is greater than 1, the recursion step generates *two* recursive calls (line 37), each of which is for a slightly simpler problem than the original call to **`Fibonacci`**. Figure 6.17 shows how method **`Fibonacci`** would evaluate **`Fibonacci(3)`**.

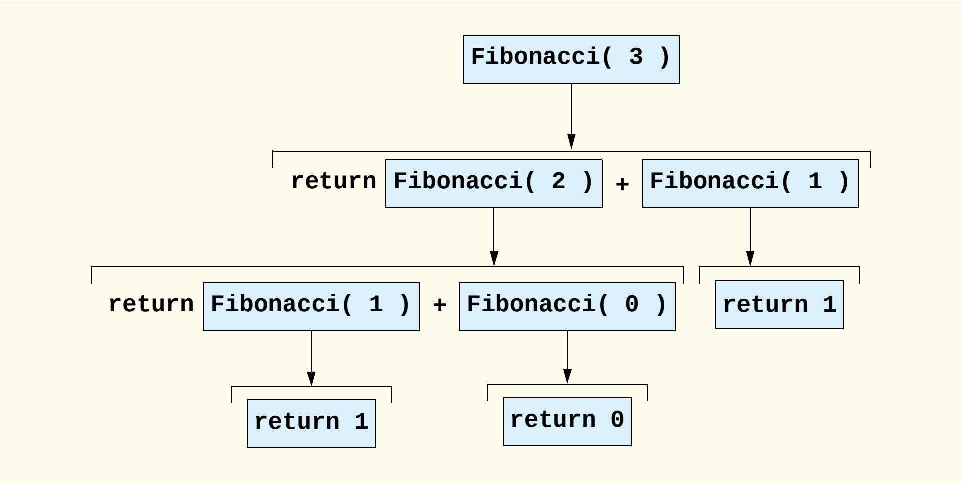

Fibonacci(3)
return Fibonacci(2) + Fibonacci(1)
return Fibonacci(1) + Fibonacci(0)
return 1
return 1
return 0

Fig. 6.17 Set of recursive calls to method **`Fibonacci`**.

This figure raises some issues about the order in which C# compilers will evaluate operands. Figure 6.17 shows that, during the evaluation of **`Fibonacci(3)`**, two recursive calls will be made—**`Fibonacci(2)`** and **`Fibonacci(1)`**. In what order will these calls be made? Most programmers assume the operands will be evaluated from left to right; in C# this is indeed true.

The C and C++ languages (on which many of C#'s features are based) do not specify the order in which the operands of most operators (including **`+`**) are evaluated. Therefore, in those languages, the programmer can make no assumption about the order in which these calls execute. The calls could, in fact, execute **`Fibonacci(2)`**, then **`Fibonacci(1)`**, or they could execute in the reverse order (**`Fibonacci(1)`**, then **`Fibonacci(2)`**). In this program and in most other programs, the final result would be the same. However, in some programs, the evaluation of an operand could have *side effects* that would affect the expression's final result. C# specifies that the order of evaluation of the operands is from left to right. Thus, the method calls are first **`Fibonacci(2)`**, then **`Fibonacci(1)`**.

Good Programming Practice 6.5

Do not write expressions that depend on the order of evaluation of the operator's operands. Doing so often results in programs that are difficult to read, debug, modify and maintain.

A word of caution about using a recursive program to generate Fibonacci numbers: each invocation of the **`Fibonacci`** method that does not match one of the base cases (i.e., 0 or 1) results in two recursive calls to the **`Fibonacci`** method. This quickly results in many method invocations. Calculating the Fibonacci value of 20 using the program in Fig. 6.16 requires 21,891 calls to the **`Fibonacci`** method; calculating the Fibonacci value of 30 requires 2,692,537 calls to the **`Fibonacci`** method.

As the programmer tries larger values, each consecutive Fibonacci number that the program is asked to calculate results in a substantial increase in the number of calls to the **`Fibonacci`** method and hence in calculation time. For example, the Fibonacci value 31 requires 4,356,617 calls, and the Fibonacci value of 32 requires 7,049,155 calls. As you can see, the number of calls to Fibonacci increases quickly—1,664,080 additional calls between the Fibonacci values of 30 and 31, and 2,692,538 additional calls between the Fibonacci values of 31 and 32. This difference in number of calls made between the Fibonacci values of 31 and 32 is more than 1.5 times the difference for Fibonacci values of 30 and 31. Problems of this nature humble even the world's most powerful computers! In the field called *complexity theory*, computer scientists determine how hard algorithms work to do their jobs. Complexity issues are discussed in detail in the upper-level computer science curriculum course generally called "Algorithms."

Performance Tip 6.1

Avoid Fibonacci-style recursive programs, which result in an exponential "explosion" of method calls.

6.16 Recursion vs. Iteration

In the previous sections, we studied two methods that can be implemented either recursively or iteratively. In this section, we compare the two approaches and discuss why the programmer might choose one approach over the other.

Both iteration and recursion are based on a control structure—iteration uses a repetition structure (such as **`for`**, **`while`** or **`do`**/**`while`**) and recursion uses a selection structure (such as **`if`**, **`if`**/**`else`** or **`switch`**). Both iteration and recursion involve repetition—iteration explicitly uses a repetition structure and recursion achieves repetition through repeated method calls. Iteration and recursion each involve a termination test—iteration terminates when the loop-continuation condition fails and recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion both gradually approach termination—iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail and recursion keeps producing simpler versions of the original problem until a base case is reached. Both iteration and recursion can execute infinitely—an infinite loop occurs with iteration if the loop-continuation test never becomes false and infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on a base case.

Recursion has disadvantages as well. It repeatedly invokes the mechanism, and consequently the overhead, of method calls. This can be costly in both processor time and memory space. Each recursive call creates another copy of the method (actually, only the method's variables); this can consume considerable memory. Iteration normally occurs within a method, so the overhead of repeated method calls and extra memory assignment is omitted. Why then would a programmer choose recursion?

Software Engineering Observation 6.11

Any problem that can be solved recursively also can be solved iteratively (nonrecursively). A recursive approach normally is chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Recursive solutions also are chosen when iterative solutions are not apparent.

Performance Tip 6.2

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 6.16

Accidentally having a nonrecursive method call itself through another method can cause infinite recursion.

Most programming textbooks introduce recursion much later than we have done in this book. We feel that recursion is a sufficiently rich and complex topic that it is better to introduce it early and spread its examples over the remainder of the text.

6.17 Method Overloading

C# enables several methods of the same name to be defined in the same class, as long as these methods have different sets of parameters (number of parameters, types of parameters or order of the parameters). This is called *method overloading*. When an overloaded method is called, the C# compiler selects the proper method by examining the number, types and order of the call's arguments. Method overloading commonly is used to create several methods with the same name that perform similar tasks, but on different data types. Figure 6.18 uses overloaded method `Square` to calculate the square of an `int` and a `double`.

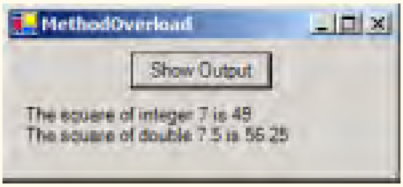
Good Programming Practice 6.6

Overloading methods that perform closely related tasks can make programs more readable and understandable.

```
// Fig. 6.18: MethodOverload.cs
// Using overloaded methods.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class MethodOverload : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button showOutputButton;
   private System.Windows.Forms.Label outputLabel;

   // Visual Studio .NET generated code

   // first version, takes one integer
   public int Square ( int x )
   {
      return x * x;
   }

   // second version, takes one double
   public double Square ( double y )
   {
      return y * y;
   }

```

Fig. 6.18 Using overloaded methods. (Part 1 of 2.)

```
29      // main entry point for the application
30      [STAThread]
31      static void Main()
32      {
33         Application.Run( new MethodOverload() );
34      }
35
36      private void showOutputButton_Click( object sender,
37         System.EventArgs e )
38      {
39         // call both versions of Square
40         outputLabel.Text =
41            "The square of integer 7 is " + Square( 7 ) +
42            "\nThe square of double 7.5 is " + Square ( 7.5 );
43      }
44
45   } // end of class MethodOverload
```



```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

Fig. 6.18 Using overloaded methods. (Part 2 of 2.)

The compiler distinguishes overloaded methods by their *signatures*. A method's signature is a combination of the method's name and parameter types. If the compiler looked only at method names during compilation, the code in Fig. 6.18 would be ambiguous—the compiler would not know how to distinguish the two **`Square`** methods. The compiler uses *overload resolution* to determine which method to call. This process first searches for all the methods that *can* be used in the context, based on the number and type of arguments that are present. It might seem that only one method would match, but recall that C# can convert variable values to other data types implicitly. Once all matching methods are found, the closest match is chosen. This match is based on a "best-fit" algorithm, which analyzes the implicit conversions that will take place.

Let us look at an example. In Fig. 6.18, the compiler might use the logical name "**`Square`** of **`int`**" for the **`Square`** method that specifies an **`int`** parameter (line 30) and "**`Square`** of **`double`**" for the **`Square`** method that specifies a **`double`** parameter (line 36). If a method **`Foo`**'s definition begins as

```
void Foo( int a, float b )
```

the compiler might use the logical name "**`Foo`** of **`int`** and **`float`**." If the parameters are specified as

```
void Foo( float a, int b )
```

the compiler might use the logical name "**`Foo`** of **`float`** and **`int`**." The order of the parameters is important to the compiler; it considers the preceding two **`Foo`** methods distinct.

So far, the logical names of methods that have been used by the compiler have not mentioned the methods' return types. This is because method calls cannot be distinguished by return type. The program in Fig. 6.19 illustrates the syntax error that is generated when two methods have the same signature and different return types. Overloaded methods with different parameter lists can have different return types. Overloaded methods need not have the same number of parameters.

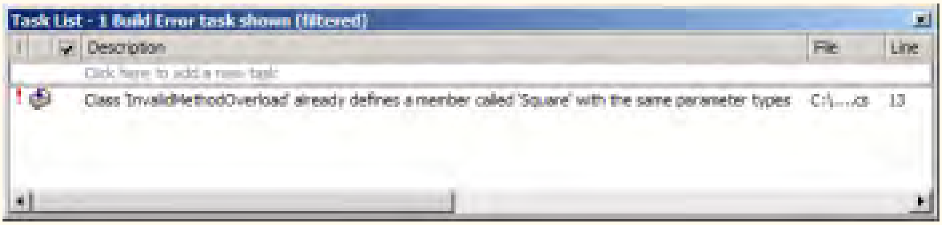
Common Programming Error 6.17

Creating overloaded methods with identical parameter lists and different return types is a syntax error.

SUMMARY

- The best way to develop and maintain a large program is to construct it from small pieces, or modules. This technique is called divide and conquer.
- Modules can be created with methods and classes.
- Programs are written by combining new methods and classes that the programmer writes with "prepackaged" methods and classes in the .NET Framework Library, and in various other method and class libraries.
- The .NET Framework Library provides a rich collection of classes and methods for performing common mathematical calculations, string manipulations, character manipulations, input/output, error checking and other useful operations.

```
// Fig. 6.19: InvalidMethodOverload.cs
// Demonstrating incorrect method overloading.

public class InvalidMethodOverload
{
   public int Square( double x )
   {
      return x * x;
   }

   // ERROR! Second Square method takes same number, order
   // and types of arguments.
   public double Square( double y )
   {
      return y * y;
   }
}
```


Task List - 1 Build Error task shown (filtered)
Description
File
Line
Click here to add a new task

```
Class 'InvalidMethodOverload' already defines a member called 'Square' with the same parameter types   C:\...cs   13
```

Fig. 6.19 Syntax error generated from overloaded methods with identical parameter lists and different return types.

- The programmer can write methods to define specific tasks that may be used at many points in a program. These methods sometimes are referred to as programmer-defined methods.
- The actual statements defining the method are written only once and are hidden from other methods.
- Methods are called by writing the name of the method (sometimes preceded by the class name and a dot operator), followed by a left parenthesis, the method's argument (or a comma-separated list of arguments) and a right parenthesis.
- All variables declared in method definitions are local variables—they are known only in the method in which they are defined.
- Packaging code as a method allows that code to be executed from several locations in a program when the method is called.
- The `return` statement in a method passes the results of the method back to the calling method.
- The format of a method definition is

```
return-value-type  method-name( parameter-list )
{
  declarations and statements
}
```

- The first line of a method definition is sometimes known as the method header. The attributes and modifiers in the method header are used to specify information about the method.
- The method *return-value-type* is the data type of the result that is returned from the method to the caller. Methods can return one value at most.
- The *parameter-list* is a comma-separated list containing the declarations of the parameters received by the called method. There must be one argument in the method call for each parameter in the method definition.
- The declarations and statements within the braces that follow the method header form the method body.
- Variables can be declared in any block, and blocks can be nested.
- A method cannot be defined inside another method.
- In many cases, an argument value that does not correspond precisely to the parameter types in the method definition is converted to the proper type before the method is called.
- When an argument is passed by value, a copy of the argument's value is made and passed to the called method.
- With pass-by-reference, the caller enables the called method to access the caller's data directly and to modify that data if the called method chooses.
- The class `Random` can be used to generate random numbers.
- An event is a signal that is sent to a program when some action takes place, such as when the user clicks a button. The programmer writes the application to perform tasks when these events occur. An event handler is a method that executes when an event occurs (or is "raised").
- An identifier's duration (its lifetime) is the period during which that identifier exists in memory.
- Identifiers that represent local variables in a method (i.e., parameters and variables declared in the method body) have automatic duration. Automatic-duration variables are created when program control reaches the variable's declaration. They exist while the block in which they are declared is active, and they are destroyed when the block in which they are declared is exited.
- The scope (sometimes called a declaration space) of an identifier for a variable, reference or method is the portion of the program in which that identifier can be referenced.

- A local variable or reference declared in a block can be used only in that block or in blocks nested within that block.
- Members of a class have class scope and are visible in what is known as the declaration space of the class.
- Class scope enables a class's methods to access directly all members defined in that class or inherited into that class. (**`static`** members are an exception to this rule.)
- Any variable declared in the initialization portion of a **`for`** structure will be visible only within that **`for`** structure.
- A recursive method is one that calls itself either directly, or indirectly through another method.
- A recursive method knows how to solve only the simplest case(s), or base case(s). If the method is called with a base case, the method returns a result. If the method is called with a more complex problem, the method divides the problem into two conceptual pieces—a piece that the method knows how to solve (base case) and a piece that the method does not know how to solve.
- To make recursion feasible, the portion of the problem that the method does not know how to solve must resemble the original problem, but be a slightly simpler or smaller version.
- Certain recursive methods can lead to an exponential "explosion" of method calls.
- Both iteration and recursion are based on a control structure. Iteration uses a repetition structure (such as **`for`**, **`while`** or **`do`**/**`while`**); recursion uses a selection structure (such as **`if`**, **`if`**/**`else`** or **`switch`**).
- Both iteration and recursion involve repetition. Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated method calls.
- Iteration and recursion each involve a termination test. Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Both iteration and recursion can execute infinitely. An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.
- A recursive approach normally is chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug.
- Several methods can have the same name, as long as these methods have different sets of parameters, in terms of number of parameters, types of the parameters and order of the parameters. This is called method overloading.
- Method overloading commonly is used to create several methods with the same name that perform similar tasks, but on different data types.

TERMINOLOGY

`.` (dot operator)
argument to a method call
automatic duration
automatic initialization of a variable
base case
`Button` class
calling method
cast operator
class
`Click` event
coercion of arguments
comma-separated list of arguments
complexity theory
constant variable
control structures in iteration
control structures in recursion
divide-and-conquer approach
duration of an identifier
event handling
exhausting memory

exponential “explosion” of calls
factorial method
Fibonacci series defined recursively
golden ratio
hierarchical structure
infinite loop
infinite recursion
instance variables of a class
invoke a method
lifetime of an identifier
local variable
method
method body
method call
method header
method overloading
modularizing a program with methods
monolithic program
named constant
nested block
nested control structure
`out` parameter
overloaded method
parameter list
pass-by-reference
pass-by-value
principle of least privilege
programmer-defined method
promotions for primitive data types
`Random` class
recursive evaluation
recursive method
`ref` parameter
`return` keyword
return-value type
scaling factor
scope of an identifier
sequence of random numbers
shifting value
side effect
signature
simulation
software reusability
static duration
termination test
user-defined method
user interface event
`void` return-value type

SELF-REVIEW EXERCISES

6.1 Fill in the blanks in each of the following statements:
a) Program modules in C# are called __________ and __________.
b) A method is invoked with a __________.
c) A variable known only within the method in which it is defined is called a __________.
d) The __________ statement in a called method can be used to pass the value of an expression back to the calling method.
e) The keyword __________ is used in a method header to indicate that a method does not return a value.
f) The __________ of an identifier is the portion of the program in which the identifier can be used.
g) The three ways to return control from a called method to a caller are __________, __________ and __________.
h) The __________ method is used to produce random numbers.
i) Variables declared in a block or in a method’s parameter list are of __________ duration.
j) A method that calls itself either directly or indirectly is a __________ method.
k) A recursive method typically has two components: one that provides a means for the recursion to terminate by testing for a __________ case, and one that expresses the problem as a recursive call for a slightly simpler problem than the original call.
l) In C#, it is possible to have various methods with the same name that operate on different types or numbers of arguments. This is called method __________.
m) Local variables declared at the beginning of a method have __________ scope, as do method parameters, which are considered local variables of the method.
n) Iteration is based on a control structure. It uses a __________ structure.
o) Recursion is based on a control structure. It uses a __________ structure.

p) Recursion achieves repetition through repeated __________ calls.
q) The best way to develop and maintain a large program is to divide it into several smaller program __________, each of which is more manageable than the original program.
r) It is possible to define methods with the same __________, but different parameter lists.
s) Recursion terminates when a __________ is reached.
t) Placing a semicolon after the right parenthesis that encloses the parameter list of a method definition is a __________ error.
u) The __________ is a comma-separated list containing the declarations of the parameters received by the called method.
v) The __________ is the data type of the result returned from a called method.
w) A __________ is a signal that is sent when some action takes place, such as a button being clicked or a value being changed.

6.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) **`Math`** method **`Abs`** rounds its parameter to the smallest integer.
b) **`Math`** method **`Exp`** is the exponential method, e^x.
c) Variable type **`float`** can be promoted to type **`double`**.
d) Variable type **`char`** cannot be promoted to type **`int`**.
e) A recursive method is one that calls itself.
f) When a method recursively calls itself, it is known as the base case.
g) 0! is equal to 1.
h) Forgetting to return a value from a recursive method when one is needed results in a syntax error.
i) Infinite recursion occurs when a method converges on the base case.
j) A recursive implementation of the **`Fibonacci`** method is always efficient.
k) Any problem that can be solved recursively also can be solved iteratively.

6.3 For the following program, state the scope (either class scope or block scope) of each of the following elements:

a) The variable **`x`**.
b) The variable **`y`**.
c) The method **`cube`**.
d) The method **`paint`**.
e) The variable **`yPos`**.

```
public class CubeTest {
   int x;

   public void paint()
   {
      int yPos = 25;

      for ( x = 1; x <= 10; x++ ) {
         Console.WriteLine( x );
         yPos += 15;
      }
   }

   public int cube( int y )
   {
      return y * y * y;
   }
}
```

6.4 Write an application that tests whether the examples of the math library method calls shown in Fig. 6.2 actually produce the indicated results.

6.5 Give the method header for each of the following methods:

a) Method **`hypotenuse`**, which takes two double-precision, floating-point arguments **`side1`** and **`side2`** and returns a double-precision, floating-point result.

b) Method **`smallest`**, which takes three integers, **`x`**, **`y`**, **`z`**, and returns an integer.

c) Method **`instructions`**, which does not take any arguments and does not return a value. [*Note*: Such methods commonly are used to display instructions to a user.]

d) Method **`intToFloat`**, which takes an integer argument, **`number`**, and returns a floating-point result.

6.6 Find the error in each of the following program segments and explain how the error can be corrected:

a)
```
int g() {
   Console.WriteLine( "Inside method g" );
   int h() {
      Console.WriteLine( "Inside method h" );
   }
}
```
b)
```
int sum( int x, int y ) {
   int result;
   result = x + y;
}
```
c)
```
int sum( int n ) {
   if ( n == 0 )
      return 0;
   else
      n + sum( n - 1 );
}
```
d)
```
void f( float a ); {
   float a;
   Console.WriteLine( a );
}
```
e)
```
void product() {
   int a = 6, b = 5, c = 4, result;
   result = a * b * c;
   Console.WriteLine( "Result is " + result );
   return result;
}
```

ANSWERS TO SELF-REVIEW EXERCISES

6.1 a) methods, classes. b) method call. c) local variable. d) **`return`**. e) **`void`**. f) scope. g) **`return;`**, **`return`** *expression***`;`**, encountering the closing right brace of a method. h) **`Random.Next`**. i) automatic. j) recursive. k) base. l) overloading. m) block. n) repetition. o) selection. p) method. q) modules. r) name. s) base case. t) syntax. u) parameter list. v) return-value-type. w) event.

6.2 a) False. **`Math`** method **`Abs`** returns the absolute value of a number. b) True. c) True. d) False. Type **`char`** can be promoted to **`int`**, **`float`**, **`long`** and **`double`**. e) True. f) False. When a method recursively calls itself, it is known as the recursive call or recursion step. g) True. h) True. i) False. Infinite recursion will occur when a recursive method does not converge on the base case. j) False. Recursion repeatedly invokes the mechanism, and consequently, the overhead, of method calls. k) True.

6.3 a) Class scope. b) Block scope. c) Class scope. d) Class scope. e) Block scope.

6.4 The following code demonstrates the use of some `Math` library method calls:

```
// Exercise 6.4: MathTest.cs
// Testing the Math class methods
using System;

public class MathTest {
   public static void Main( string[] args )
   {
      Console.WriteLine( "Math.Abs( 23.7 ) = " +
                          Math.Abs( 23.7 ) );
      Console.WriteLine( "Math.Abs( 0.0 ) = " +
                          Math.Abs( 0.0 ) );
      Console.WriteLine( "Math.Abs( -23.7 ) = " +
                          Math.Abs( -23.7 ) );
      Console.WriteLine( "Math.Ceiling( 9.2 ) = " +
                          Math.Ceiling( 9.2 ) );
      Console.WriteLine( "Math.Ceiling( -9.8 ) = " +
                          Math.Ceiling( -9.8 ) );
      Console.WriteLine( "Math.Cos( 0.0 ) = " +
                          Math.Cos( 0.0 ) );
      Console.WriteLine( "Math.Exp( 1.0 ) = " +
                          Math.Exp( 1.0 ) );
      Console.WriteLine( "Math.Exp( 2.0 ) = " +
                          Math.Exp( 2.0 ) );
      Console.WriteLine( "Math.Floor( 9.2 ) = " +
                          Math.Floor( 9.2 ) );
      Console.WriteLine( "Math.Floor( -9.8 ) = " +
                          Math.Floor( -9.8 ) );
      Console.WriteLine( "Math.Log( 2.718282 ) = " +
                          Math.Log( 2.718282 ) );
      Console.WriteLine( "Math.Log( 7.389056 ) = " +
                          Math.Log( 7.389056 ) );
      Console.WriteLine( "Math.Max( 2.3, 12.7 ) = " +
                          Math.Max( 2.3, 12.7 ) );
      Console.WriteLine( "Math.Max( -2.3, -12.7 ) = " +
                          Math.Max( -2.3, -12.7 ) );
      Console.WriteLine( "Math.Min( 2.3, 12.7 ) = " +
                          Math.Min( 2.3, 12.7 ) );
      Console.WriteLine( "Math.Min( -2.3, -12.7 ) = " +
                          Math.Min( -2.3, -12.7 ) );
      Console.WriteLine( "Math.Pow( 2, 7 ) = " +
                          Math.Pow( 2, 7 ) );
      Console.WriteLine( "Math.Pow( 9, .5 ) = " +
                          Math.Pow( 9, .5 ) );
      Console.WriteLine( "Math.Sin( 0.0 ) = " +
                          Math.Sin( 0.0 ) );
      Console.WriteLine( "Math.Sqrt( 25.0 ) = " +
                          Math.Sqrt( 25.0 ) );
      Console.WriteLine( "Math.Tan( 0.0 ) = " +
                          Math.Tan( 0.0 ) );
   }
}
```

```
Math.Abs( 23.7 ) = 23.7
Math.Abs( 0.0 ) = 0
Math.Abs( -23.7 ) = 23.7
Math.Ceiling( 9.2 ) = 10
Math.Ceiling( -9.8 ) = -9
Math.Cos( 0.0 ) = 1
Math.Exp( 1.0 ) = 2.71828
Math.Exp( 2.0 ) = 7.38906
Math.Floor( 9.2 ) = 9
Math.Floor( -9.8 ) = -10
Math.Log( 2.718282 ) = 1
Math.Log( 7.389056 ) = 2
Math.Max( 2.3, 12.7 ) = 12.7
Math.Max( -2.3, -12.7 ) = -2.3
Math.Min( 2.3, 12.7 ) = 2.3
Math.Min( -2.3, -12.7 ) = -12.7
Math.Pow( 2, 7 ) = 128
Math.Pow( 9, .5 ) = 3
Math.Sin( 0.0 ) = 0
Math.Sqrt( 25.0 ) = 5
Math.Tan( 0.0 ) = 0
```

6.5 a) `double hypotenuse( double side1, double side2 )`
b) `int smallest( int x, int y, int z )`
c) `void instructions()`
d) `float intToFloat( int number )`

6.6 a) Error: Method **`h`** is defined in method **`g`**.
Correction: Move the definition of **`h`** out of the definition of **`g`**.

b) Error: The method is supposed to return an integer, but does not.
Correction: Delete variable **`result`** and place the following statement in the method:

```
return x + y;
```

or add the following statement at the end of the method body:

```
return result;
```

c) Error: The result of **`n`** + **`sum(n - 1)`** is not returned by this recursive method, resulting in a syntax error.
Correction: Rewrite the statement in the **`else`** clause as

```
return n + sum( n - 1 );
```

d) Error: The semicolon after the right parenthesis that encloses the parameter list, and the redefining of the parameter **`a`** in the method definition are both incorrect.
Correction: Delete the semicolon after the right parenthesis of the parameter list and delete the declaration **`float a;`**.

e) Error: The method returns a value when it is not supposed to.
Correction: Change the return type to **`int`**.

EXERCISES

6.7 What is the value of **`x`** after each of the following statements is performed?
a) `x = Math.Abs( 7.5 );`
b) `x = Math.Floor( 7.5 );`
c) `x = Math.Abs( 0.0 );`
d) `x = Math.Ceiling( 0.0 );`

e) `x = Math.Abs( -6.4 );`
f) `x = Math.Ceiling( -6.4 );`
g) `x = Math.Ceiling( -Math.Abs( -8 + Math.Floor( -5.5 ) ) );`

6.8 A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 per hour for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write a program that calculates and displays the parking charges for each customer who parked a car in this garage yesterday. You should enter in a **TextBox** the hours parked for each customer. The program should display the charge for the current customer. The program should use the method **CalculateCharges** to determine the charge for each customer. Use the techniques described in the chapter to read the double value from a **TextBox**.

6.9 Write a method **IntegerPower(base, exponent)** that returns the value of

$\mathtt{base}^{\mathtt{exponent}}$

For example, **IntegerPower(3, 4) = 3 * 3 * 3 * 3**. Assume that **exponent** is a positive, nonzero integer, and **base** is an integer. Method **IntegerPower** should use **for** or **while** to control the calculation. Do not use any **Math** library methods. Incorporate this method into a Windows application that reads integer values from **TextBox**es for **base** and **exponent** from the user and performs the calculation with the **IntegerPower** method.

6.10 Define a method **Hypotenuse** that calculates the length of the hypotenuse of a right triangle when the other two sides are given. The method should take two arguments of type **double** and return the hypotenuse as a **double**. Incorporate this method into a Windows application that reads integer values for **side1** and **side2** from **TextBox**es and performs the calculation with the **Hypotenuse** method. Determine the length of the hypotenuse for each of the following triangles:

Triangle	Side 1	Side 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

6.11 Write a method **SquareOfAsterisks** that displays a solid square of asterisks whose side is specified in integer parameter **side**. For example, if **side** is **4**, the method displays

```
****
****
****
****
```

Incorporate this method into an application that reads an integer value for **side** from the user and performs the drawing with the **SquareOfAsterisks** method. This method should gather data from **Textbox**es and should print to a **label**.

6.12 Modify the method created in Exercise 6.11 to form the square out of whatever character is contained in character parameter **fillCharacter**. Thus, if **side** is **5** and **fillCharacter** is “**#**,” this method should print

```
#####
#####
#####
#####
#####
```

6.13 Write an application that simulates coin tossing. Let the program toss the coin each time the user presses the “**`Toss`**” button. Count the number of times each side of the coin appears. Display the results. The program should call a separate method **`Flip`** that takes no arguments and returns **`false`** for tails and **`true`** for heads. [*Note*: If the program realistically simulates the coin tossing, each side of the coin should appear approximately half of the time.]

6.14 Computers are playing an increasing role in education. Write a program that will help an elementary school student learn multiplication. Use the **`Next`** method from an object of type **`Random`** to produce two positive one-digit integers. It should display a question in the status bar, such as

```
How much is 6 times 7?
```

The student should then type the answer into a **`TextBox`**. Your program should check the student’s answer. If it is correct, draw the string **`"Very good!"`** in a read-only **`TextBox`**, then ask another multiplication question. If the answer is wrong, draw the string **`"No. Please try again."`** in the same read-only **`TextBox`**, then let the student try the same question again until the student finally gets it right. A separate method should be used to generate each new question. This method should be called once when the program begins execution and each time the user answers the question correctly.

6.15 (*Towers of Hanoi*) Every budding computer scientist must grapple with certain classic problems and the Towers of Hanoi (Fig. 6.20) is one of the most famous. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time, and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

Let us assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would find ourselves hopelessly knotted up in managing the disks. However, if we attack the problem with recursion in mind, it becomes tractable. Moving n disks can be viewed in terms of moving only $n-1$ disks (and hence, the recursion) as follows:

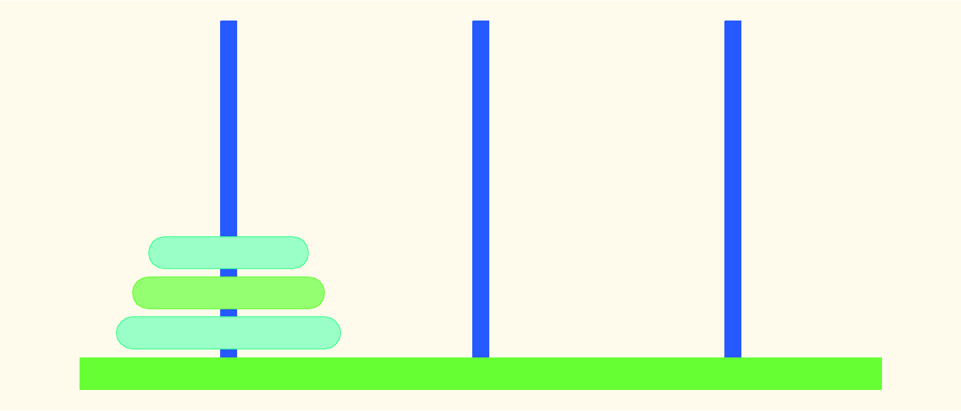


Fig. 6.20 The Towers of Hanoi for the case with four disks.

a) Move *n* – 1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
b) Move the last disk (the largest) from peg 1 to peg 3.
c) Move the *n* – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving *n* = 1 disk (i.e., the base case). This is accomplished by trivially moving the disk without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Allow the user to enter the number of disks in a **`TextBox`**. Use a recursive **`Tower`** method with four parameters:

a) The number of disks to be moved
b) The peg on which these disks are threaded initially
c) The peg to which this stack of disks is to be moved
d) The peg to be used as a temporary holding area

Your program should display in a read-only **`TextBox`** with scrolling functionality the precise instructions for moving the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

1 → 3 (This means move one disk from peg 1 to peg 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

6.16 The *greatest common divisor* of integers **`x`** and **`y`** is the largest integer that evenly divides both **`x`** and **`y`**. Write a recursive method **`Gcd`** that returns the greatest common divisor of **`x`** and **`y`**. The **`Gcd`** of **`x`** and **`y`** is defined recursively as follows: If **`y`** is equal to **`0`**, then **`Gcd( x, y )`** is **`x`**; otherwise, **`Gcd( x, y )`** is **`Gcd( y, x % y )`**, where **`%`** is the modulus operator.

7

Arrays

Objectives

- To introduce the array data structure.
- To understand how arrays store, sort and search lists and tables of values.
- To understand how to declare an array, initialize an array and refer to individual elements of an array.
- To be able to pass arrays to methods.
- To understand basic sorting techniques.
- To be able to declare and manipulate multiple-subscript arrays.

With sobs and tears he sorted out
Those of the largest size …
Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
Robert Herrick

Now go, write it before them in a table,
and note it in a book.
Isaiah 30:8

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
William Shakespeare

Outline

7.1 Introduction
7.2 Arrays
7.3 Declaring and Allocating Arrays
7.4 Examples Using Arrays
    7.4.1 Allocating an Array and Initializing Its Elements
    7.4.2 Totaling the Elements of an Array
    7.4.5 Using Arrays to Analyze Survey Results
    7.4.3 Using Histograms to Display Array Data Graphically
    7.4.4 Using the Elements of an Array as Counters
    7.4.5 Using Arrays to Analyze Survey Results
7.5 Passing Arrays to Methods
7.6 Passing Arrays by Value and by Reference
7.7 Sorting Arrays
7.8 Searching Arrays: Linear Search and Binary Search
    7.8.1 Searching an Array with Linear Search
    7.8.2 Searching a Sorted Array with Binary Search
7.9 Multiple-Subscripted Arrays
7.10 `foreach` Repetition Structure

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

7.1 Introduction

This chapter serves as an introduction to data structures. *Arrays* are data structures consisting of data items of the same type. Arrays are “static” entities, in that they remain the same size once they are created. We begin by learning about creating and accessing arrays, then use this knowledge to begin more complex manipulations of arrays, including powerful searching and sorting techniques. We then demonstrate creating more sophisticated arrays that have multiple dimensions. Chapter 24, Data Structures, introduces dynamic data structures such as lists, queues, stacks and trees that can grow and shrink as programs execute. We also introduce C#’s predefined data structures that enable the programmer to use existing data structures for lists, queues, stacks and trees, rather than having to “reinvent the wheel.”

7.2 Arrays

An array is a group of contiguous memory locations that all have the same name and type. To refer to a particular location or element in the array, we specify the name of the array and the *position number* (a value that indicates a specific location within the array) of the element to which we refer.

Figure 7.1 shows an integer array called **`c`**. This array contains 12 *elements.* A program can refer to any element of an array by giving the name of the array followed by the position number of the element in square brackets (**`[]`**). The first element in every array is the *zeroth element*. Thus, the first element of array **`c`** is referred to as **`c[ 0 ]`**, the second element of array **`c`** is referred to as **`c[ 1 ]`**, the seventh element of array **`c`** is referred to as **`c[ 6 ]`** and so on. The *i*th element of array **`c`** is referred to as **`c[ i - 1 ]`**. Array names follow the same conventions as other variable names, as discussed in Chapter 3, Introduction to C# Programming.

The position number in square brackets is more formally called a *subscript* (or an *index*). A subscript must be an integer or an integer expression. If a program uses an expression as a subscript, the program evaluates the expression first to determine the subscript. For example, if variable **`a`** is equal to **`5`** and variable **`b`** is equal to **`6`**, then the statement

```
c[ a + b ] += 2;
```

adds 2 to array element **`c[ 11 ]`**. Note that a subscripted array name is an *lvalue*—it can be used on the left side of an assignment to place a new value into an array element.

Let us examine array c in Fig. 7.1 more closely. The name of the array is c. Every array in C# "knows" its own length. The length of the array is determined by the expression:

```
c.Length
```

The array's 12 elements are referred to as **`c[ 0 ]`**, **`c[ 1 ]`**, **`c[ 2 ]`**, …, **`c[ 11 ]`**. The *value* of **`c[ 0 ]`** is **`-45`**, the value of **`c[ 1 ]`** is **`6`**, the value of **`c[ 2 ]`** is **`0`**, the value of **`c[ 7 ]`** is **`62`** and the value of **`c[ 11 ]`** is **`78`**. To calculate the sum of the values contained in the first three elements of array **`c`** and to store the result in variable **`sum`**, we would write

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

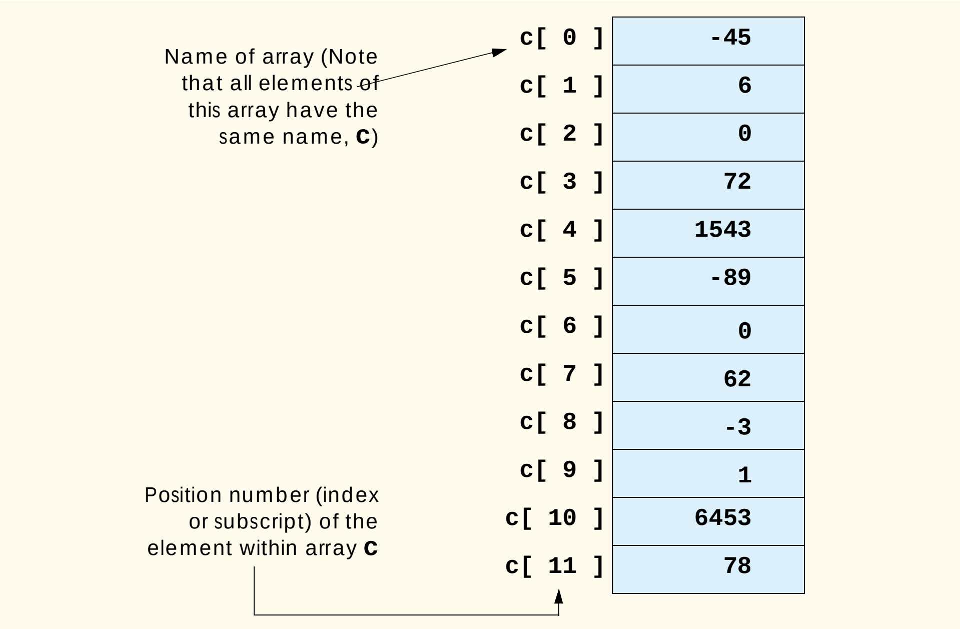

Name of array (Note that all elements of this array have the same name, c)
c[0]
-45
c[1]
6
c[2]
0
c[3]
72
c[4]
1543
c[5]
-89
c[6]
0
c[7]
62
c[8]
-3
c[9]
1
c[10]
6453
c[11]
78
Position number (index or subscript) of the element within array c

Fig. 7.1 A 12-element array.

To divide the value of the seventh element of array **`c`** by **`2`** and assign the result to the variable **`x`**, we would write

```
x = c[ 6 ] / 2;
```

Common Programming Error 7.1

It is important to note the difference between the "seventh element of the array" and "array element seven." Array subscripts begin at 0, thus the "seventh element of the array" has a subscript of 6, while "array element seven" has a subscript of 7 and is actually the eighth element of the array. This confusion is a source of "off-by-one" errors.

The brackets that enclose the subscript of an array are operators. Brackets have the same level of precedence as parentheses. The chart in Fig. 7.2 shows the precedence and associativity of the operators introduced to this point in the text. They are displayed top to bottom in decreasing order of precedence, with their associativity and type. The reader should note that the **`++`** and **`--`** operators in the first row represent the postincrement and postdecrement operators, while the **`++`** and **`--`** operators in the second row represent the preincrement and predecrement operators. Also, notice that in the first row the associativity is mixed. This is because the associativity of the postincrement and postdecrement operators is right to left, while the associativity for the other operators is left to right.

7.3 Declaring and Allocating Arrays

Arrays occupy space in memory. The programmer specifies the type of the elements and uses operator **`new`** to allocate dynamically the number of elements required by each array. Arrays are allocated with **`new`** because arrays are objects and all objects must be created with **`new`**. We will see an exception to this rule shortly.

Operators	**Associativity**	**Type**
`()  []  .  ++  --`	left to right	highest (unary postfix)
`++  --  +  -  !` `(`*type*`)`	right to left	unary (unary prefix)
`*   /   %`	left to right	multiplicative
`+   -`	left to right	additive
`<  <=  >  >=`	left to right	relational
`==  !=`	left to right	equality
`&`	left to right	logical AND
`^`	left to right	logical exclusive OR
`|`	left to right	logical inclusive OR
`&&`	left to right	conditional AND
`||`	left to right	conditional OR
`?:`	right to left	conditional
`=  +=  -=  *=  /=  %=`	right to left	assignment

Fig. 7.2 Precedence and associativity of the operators discussed so far.

The declaration

```
int[] c = new int[ 12 ];
```

allocates 12 elements for integer array **`c`**. The preceding statement can also be performed in two steps as follows:

```
int[] c;                // declares the array
c = new int[ 12 ];  // allocates the reference to the array
```

When arrays are allocated, the elements are initialized to zero for the numeric primitive-data-type variables, to **`false`** for **`bool`** variables and to **`null`** for reference types.

Common Programming Error 7.2

Unlike in C or C++, in C# the number of elements in the array is never specified in the square brackets after the array name. The declaration **`int[ 12 ] c;`** *causes a syntax error.*

Memory may be reserved for several arrays with a single declaration. The following declaration reserves 100 elements for **`string`** array **`b`** and 27 elements for **`string`** array **`x`**:

```
string[] b = new string[ 100 ], x = new string[ 27 ];
```

Similarly, the following declaration reserves 10 elements for **`array1`** and 20 elements for **`array2`** (both of type **`double`**):

```
double[] array1 = new double[ 10 ],
         array2 = new double[ 20 ];
```

Arrays may be declared to contain most data types. In an array of value types, every element of the array contains one value of the declared type. For example, every element of an **`int`** array is an **`int`** value.

In an array of reference types, every element of the array is a reference to an object of the data type of the array. For example, every element of a **`string`** array is a reference to a **`string`**. Each of these **`string`** references has the value **`null`** by default.

7.4 Examples Using Arrays

This section presents several examples using arrays that demonstrate declaring arrays, allocating arrays, initializing arrays and manipulating array elements in various ways. For simplicity, the examples in this section use arrays that contain elements of type **`int`**. Please remember that a program can declare arrays of most data types.

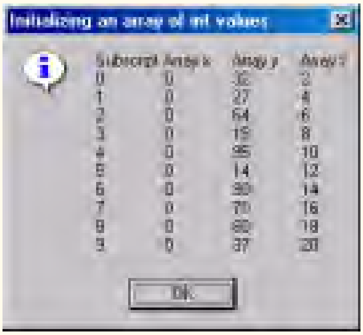
7.4.1 Allocating an Array and Initializing Its Elements

Figure 7.3 creates three integer arrays of 10 elements and displays those arrays in tabular format. The program demonstrates several techniques for declaring and initializing arrays.

```
// Fig 7.3: InitArray.cs
// Different ways of initializing arrays.

using System;
using System.Windows.Forms;
```

Fig. 7.3 Initializing element arrays in three different ways. (Part 1 of 2.)

```

class InitArray
{
   // main entry point for application
   static void Main( string[] args )
   {
      string output = "";

      int[] x;                    // declare reference to an array
      x = new int[ 10 ];          // dynamically allocate array and set
                                  // default values

      // initializer list specifies number of elements
      // and value of each element
      int[] y = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      const int ARRAY_SIZE = 10; // named constant
      int[] z;                   // reference to int array

      // allocate array of ARRAY_SIZE (i.e., 10) elements
      z = new int[ ARRAY_SIZE ];

      // set the values in the array
      for ( int i = 0; i < z.Length; i++ )
         z[ i ] = 2 + 2 * i;

      output += "Subscript\tArray x\tArray y\tArray z\n";

      // output values for each array
      for ( int i = 0; i < ARRAY_SIZE; i++ )
         output += i + "\t" + x[ i ] + "\t" + y[ i ] +
            "\t" + z[ i ] + "\n";

      MessageBox.Show( output,
         "Initializing an array of int values",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end Main

} // end class InitArray
```



```
Subscript  Array x  Array y  Array z
0          0        32       2
1          0        27       4
2          0        64       6
3          0        18       8
4          0        95       10
5          0        14       12
6          0        90       14
7          0        70       16
8          0        60       18
9          0        37       20
```

Fig. 7.3 Initializing element arrays in three different ways. (Part 2 of 2.)

Line 14 declares **`x`** as a reference to an array of integers. Each element in the array is of type **`int`**. The variable **`x`** is of type **`int[]`**, which denotes an array whose elements are of type **`int`**. Line 15 allocates the 10 elements of the array with **`new`** and assigns the array to reference **`x`**. Each element of this array has the default value 0.

Line 20 creates another **`int`** array and initializes each element using an *initializer list*. In this case, the number of elements in the initializer list determines the array's size. For example, line 20 creates a 10-element array with the indices **`0`**–**`9`** and the values **`32`**, **`27`**, **`64`**, and so on. Note that this declaration does not require the **`new`** operator to create the array object—the compiler allocates memory for the object when it encounters an array declaration that includes an initializer list.

On line 22, we create constant integer **`ARRAY_SIZE`** using keyword **`const`**. A constant must be initialized in the same statement where it is declared and cannot be modified thereafter. If an attempt is made to modify a **`const`** variable after it is declared, the compiler issues a syntax error.

Constants also are called *named constants*. They often are used to make a program more readable and are usually denoted with variable names in all capital letters.

Common Programming Error 7.3

Assigning a value to a constant after the variable has been initialized is a compiler error.

On lines 23 and 26, we create integer array **`z`** of length 10 using the **`ARRAY_SIZE`** named constant. The **`for`** structure in lines 29–30 initializes each element in array **`z`**. The values are generated by multiplying each successive value of the loop counter by **`2`** and adding **`2`** to the product. After this initialization, array **`z`** contains the even integers **`2`**, **`4`**, **`6`**, …, **`20`**. The **`for`** structure in lines 35–37 uses the values in arrays **`x`**, **`y`** and **`z`** to build an output string, which will be displayed in a **`MessageBox`**. Zero-based counting (remember, array subscripts start at 0) allows the loop to access every element of the array. The constant **`ARRAY_SIZE`** in the **`for`** structure condition (line 29) specifies the arrays' lengths.

7.4.2 Totaling the Elements of an Array

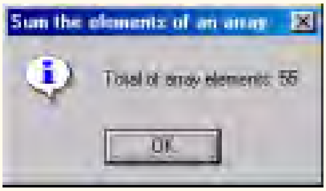
Often, the elements of an array represent series of values to be used in calculations. For example, if the elements of an array represent the grades for an exam in a class, the professor may wish to total the elements of an array, then calculate the class average for the exam.

The application in Fig. 7.4 sums the values contained in the 10-element integer array **`a`** (declared, allocated and initialized on line 12). Line 16 in the body of the **`for`** loop performs the addition using the array element at position **`i`** during each loop iteration. Note that the values being supplied as initializers for array **`a`** normally would be read into the program. For example, in a Windows application, the user could enter the values through a **`TextBox`**, or the values could be read from a file on disk. (See Chapter 17, Files and Streams.)

```
1  // Fig. 7.4: SumArray.cs
2  // Computing the sum of the elements in an array.
3
4  using System;
5  using System.Windows.Forms;
```

Fig. 7.4 Computing the sum of the elements of an array. (Part 1 of 2.)

```

class SumArray
{
   // main entry point for application
   static void Main( string[] args )
   {
      int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int total = 0;

      for ( int i = 0; i < a.Length; i++ )
         total += a[ i ];

      MessageBox.Show( "Total of array elements: " + total,
         "Sum the elements of an array",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end Main

} // end class SumArray
```



Sum the elements of an array

Total of array elements: 55

Fig. 7.4 Computing the sum of the elements of an array. (Part 2 of 2.)

7.4.3 Using Histograms to Display Array Data Graphically

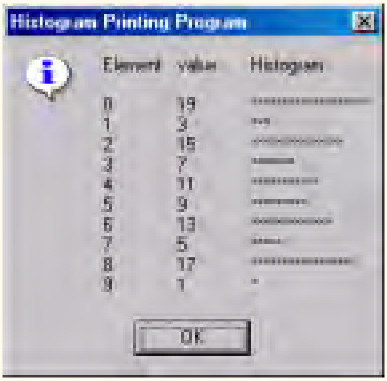
Many programs present data to users in a graphical manner. For example, numeric values often are displayed as bars in a bar chart. In such a chart, longer bars represent larger numeric values. One simple way to display numeric data graphically is with a *histogram* that shows each numeric value as a bar of asterisks (*).

Our next application (Fig. 7.5) reads numbers from an array and graphs the information in the form of a bar chart, or histogram. The program displays each number followed by a bar consisting of a corresponding number of asterisks. The nested **`for`** loops (lines 18–24) append the bars to the **`string`** that will be displayed in the **`MessageBox`**. Note the loop continuation condition of the inner **`for`** structure on line 22 (**`j <= n[ i ]`**). Each time the program reaches the inner **`for`** structure, the loop counts from **`1`** to **`n[ i ]`**, using a value in array **`n`** to determine the final value of the control variable **`j`** and the number of asterisks to display.

```
1  // Fig. 7.6: Histogram.cs
2  // Using data to create a histogram.
3
4  using System;
5  using System.Windows.Forms;
```

Fig. 7.5 Program that prints histograms. (Part 1 of 2.)

```

class Histogram
{
   // main entry point for application
   static void Main( string[] args )
   {
      int[] n = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
      string output = "";

      output += "Element\tvalue\tHistogram\n";

      // build output
      for ( int i = 0; i < n.Length; i++ )
      {
         output += "\n" + i + "\t" + n[ i ] + "\t";

         for ( int j = 1; j <= n[ i ]; j++ ) // print a bar
            output += "*";
      }

      MessageBox.Show( output, "Histogram Printing Program",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end Main

} // end class Histogram
```



```
Element	value	Histogram

0	19	*******************
1	3	***
2	15	***************
3	7	*******
4	11	***********
5	9	*********
6	13	*************
7	5	*****
8	17	*****************
9	1	*
```

Fig. 7.5 Program that prints histograms. (Part 2 of 2.)

7.4.4 Using the Elements of an Array as Counters

Sometimes programs use a series of counter variables to summarize data, such as the results of a survey. In Chapter 6, Methods, we used a series of counters in our dice-rolling program to track the number of occurrences of each side on a six-sided die as the program rolled 12 dice at a time. We also indicated that there is a more elegant method than that in Fig. 6.11 for writing the dice-rolling program. An array version of this application is shown in Fig. 7.6.

The program uses the seven-element array frequency to count the occurrences of each side of the die. Line 94, which uses the random `face` value as the subscript for array `fre-`

`quency` to determine which element should be incremented during each iteration of the loop, replaces lines 95–115 of Fig. 6.11. The random number calculation on line 88 produces numbers 1–6 (the values for a six-sided die); thus, the **`frequency`** array must be large enough to allow subscript values of 1–6. The smallest number of elements required for an array to have these subscript values is seven elements (subscript values 0–6). In this program, we ignore element 0 of array **`frequency`**. Lines 75–80 replace lines 69–81 from Fig. 6.11. We can loop through array **`frequency`**; therefore, we do not have to enumerate each line of text to display in the **`Label`**, as we did in Fig. 6.11.

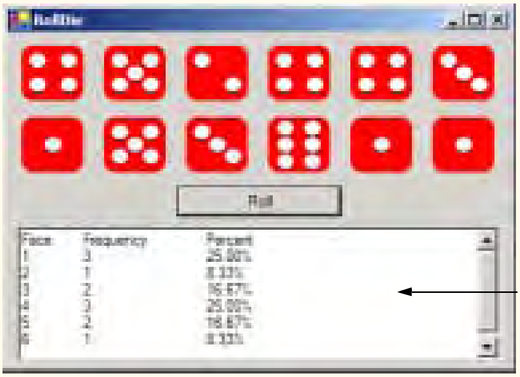
```
 1  // Fig. 7.7: RollDie.cs
 2  // Rolling 12 dice.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10  using System.IO;
11
12  public class RollDie : System.Windows.Forms.Form
13  {
14     private System.Windows.Forms.Button rollButton;
15
16     private System.Windows.Forms.RichTextBox displayTextBox;
17
18     private System.Windows.Forms.Label dieLabel1;
19     private System.Windows.Forms.Label dieLabel2;
20     private System.Windows.Forms.Label dieLabel3;
21     private System.Windows.Forms.Label dieLabel4;
22     private System.Windows.Forms.Label dieLabel5;
23     private System.Windows.Forms.Label dieLabel6;
24     private System.Windows.Forms.Label dieLabel7;
25     private System.Windows.Forms.Label dieLabel8;
26     private System.Windows.Forms.Label dieLabel9;
27     private System.Windows.Forms.Label dieLabel10;
28     private System.Windows.Forms.Label dieLabel11;
29     private System.Windows.Forms.Label dieLabel12;
30
31     private System.ComponentModel.Container components = null;
32
33     Random randomNumber = new Random();
34     int[] frequency = new int[ 7 ];
35
36     public RollDie()
37     {
38        InitializeComponent();
39     }
40
41     // Visual Studio .NET generated code
42
```

Fig. 7.6 Using arrays to eliminate a **`switch`** structure. (Part 1 of 3.)

```
   [STAThread]
   static void Main()
   {
      Application.Run( new RollDie() );
   }

   private void rollButton_Click(
      object sender, System.EventArgs e )
   {
      // pass the labels to a method that will
      // randomly assign a face to each die
      DisplayDie( dieLabel1 );
      DisplayDie( dieLabel2 );
      DisplayDie( dieLabel3 );
      DisplayDie( dieLabel4 );
      DisplayDie( dieLabel5 );
      DisplayDie( dieLabel6 );
      DisplayDie( dieLabel7 );
      DisplayDie( dieLabel8 );
      DisplayDie( dieLabel9 );
      DisplayDie( dieLabel10 );
      DisplayDie( dieLabel11 );
      DisplayDie( dieLabel12 );

      double total = 0;

      for ( int i = 1; i < 7; i++ )
         total += frequency[ i ];

      displayTextBox.Text = "Face\tFrequency\tPercent\n";

      // output frequency values
      for ( int x = 1; x < frequency.Length; x++ )
      {
         displayTextBox.Text += x + "\t" +
            frequency[ x ] + "\t\t" + String.Format( "{0:N}",
            frequency[ x ] / total * 100 ) + "%\n";
      }

   } // end Main

   // simulates roll, display proper
   // image and increment frequency
   public void DisplayDie( Label dieLabel )
   {
      int face = randomNumber.Next( 1, 7 );

      dieLabel.Image = Image.FromFile(
         Directory.GetCurrentDirectory() +
         "\\images\\die" + face + ".gif" );

      frequency[ face ]++;
   }
```

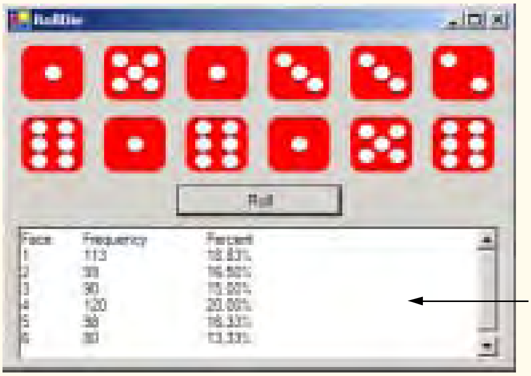
Fig. 7.6 Using arrays to eliminate a `switch` structure. (Part 2 of 3.)

```

} // end class RollDie
```



```
Face  Frequency  Percent
1     3          25.00%
2     1          8.33%
3     2          16.67%
4     3          25.00%
5     2          16.67%
6     1          8.33%
```

Results after one roll



```
Face  Frequency  Percent
1     113        18.83%
2     99         16.50%
3     90         15.00%
4     120        20.00%
5     98         16.33%
6     80         13.33%
```

Results after fifty rolls

Fig. 7.6 Using arrays to eliminate a **`switch`** structure. (Part 3 of 3.)

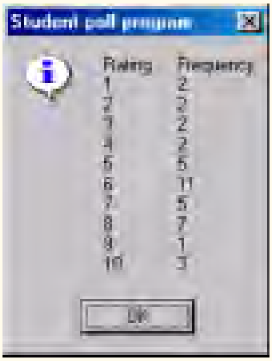
7.4.5 Using Arrays to Analyze Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the following problem statement:

> *Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10, with 1 being awful and 10 being excellent. Place the 40 responses in an integer array and summarize the frequency for each rating.*

This is a typical array processing application (Fig. 7.7). We wish to summarize the number of responses of each type (i.e., 1–10). The array **`responses`** is a 40-element integer array of the students' responses to the survey. We use an 11-element array **`frequency`** to count the number of occurrences of each response. We ignore the first element, **`frequency[ 0 ]`**, because it is more logical to have a response of **`1`** increment **`frequency[ 1 ]`** than **`frequency[ 0 ]`**. We can use each response directly as a subscript on the **`frequency`** array. Each element of the array is used as a counter for one of the survey responses.

```
// Fig. 7.5: StudentPoll.cs
// A student poll program.

using System;
using System.Windows.Forms;

class StudentPoll
{
   // main entry point for application
   static void Main( string[] args )
   {
      int[] responses = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1,
         6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
         5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

      int[] frequency = new int[ 11 ];
      string output = "";

      // increment the frequency for each response
      for ( int answer = 0; answer < responses.Length; answer++ )
         ++frequency[ responses[ answer ] ];

      output += "Rating\tFrequency\n";

      // output results
      for ( int rating = 1; rating < frequency.Length; rating++ )
         output += rating + "\t" + frequency[ rating ] + "\n";

      MessageBox.Show( output, "Student poll program",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method Main

} // end class StudentPoll
```



```
Rating  Frequency
1       2
2       2
3       2
4       2
5       5
6       11
7       5
8       7
9       1
10      3
```

Fig. 7.7 Simple student-poll analysis program.

Good Programming Practice 7.1

Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time for writing clearer programs.

The `for` loop (lines 20–21) takes the responses from the array `response` one at a time and increments one of the 10 counters in the `frequency` array (`frequency[ 1 ]` to `frequency[ 10 ]`). The key statement in the loop is on line 21, which increments the appropriate counter in the `frequency` array, depending on the value of element `responses[ answer ]`.

Let us consider several iterations of the `for` loop. When counter `answer` is `0`, `responses[ answer ]` is the value of the first element of array `responses` (i.e., `1`). In this case, the program interprets `++frequency[ responses[ answer ] ];` as `++frequency[ 1 ];`, which increments array element one. In evaluating the expression, start with the value in the innermost set of square brackets (`answer`). Once you know the value of `answer`, plug that value into the expression and evaluate the next outer set of square brackets (`responses[ answer ]`). Use that value as the subscript for the `frequency` array to determine which counter to increment.

When `answer` is `1`, `responses[ answer ]` is the value of the second element of array `responses` (i.e., `2`), so the program interprets

```
++frequency[ responses[ answer ] ];
```

as `++frequency[ 2 ];`, which increments array element two (the third element of the array). When `answer` is `2`, `responses[ answer ]` is the value of the third element of array `responses` (i.e., `6`), so the program interprets

```
++frequency[ responses[ answer ] ];
```

as `++frequency[ 6 ];`, which increments array element six (the seventh element of the array) and so on. Note that, regardless of the number of responses processed in the survey, only an 11-element array is required (ignoring element zero) to summarize the results, because all the response values are between 1 and 10, and the subscript values for an 11-element array are 0–10. The results are correct, because the elements of the `frequency` array were initialized to zero when the array was allocated with `new`.

If the data contained invalid values, such as 13, the program would attempt to add `1` to `frequency[ 13 ]`. This is outside the bounds of the array. In the C and C++ programming languages, no checks are performed to prevent programs from reading data outside the bounds of arrays. At execution time, the program would "walk" past the end of the array to where element number 13 would be located and add 1 to whatever data are stored at that location in memory. This could potentially modify another variable in the program or even result in premature program termination. The .NET framework provides mechanisms to prevent accessing elements outside the bounds of arrays.

Testing and Debugging Tip 7.1

When a C# program executes, array element subscripts are checked for validity (i.e., all subscripts must be greater than or equal to 0 and less than the length of the array).

Testing and Debugging Tip 7.2

Exceptions indicate when errors occur in programs. Programmers can write code to recover from exceptions and continue program execution instead of terminating the program abnormally. When an invalid array reference occurs, C# generates an ***`IndexOutOfRangeException`*** *exception. We discuss exceptions in more detail in Chapter 11, Exception Handling.*

Common Programming Error 7.4

Referring to an element outside the array bounds is a logic error.

Testing and Debugging Tip 7.3

When looping through an array, the array subscript never should go below 0 and should always be less than the total number of elements in the array (one less than the `length` *of the array). The loop-terminating condition should prevent accessing elements outside this range.*

Testing and Debugging Tip 7.4

Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.

7.5 Passing Arrays to Methods

To pass an array argument to a method, specify the name of the array without using brackets. For example, if array `hourlyTemperatures` declared as

```
int[] hourlyTemperatures = new int[ 24 ];
```

the method call

```
ModifyArray( hourlyTemperatures );
```

passes array `hourlyTemperatures` to method `ModifyArray`. Every array object "knows" its own size (via the `Length` instance variable), so when we pass an array object into a method, we do not pass the size of the array as an argument separately.

Although entire arrays are passed by reference, individual array elements of primitive data types are passed by value, the same way as simple variables are. (The objects referred to by individual elements of a nonprimitive-type array are still passed by reference.) Such simple single pieces of data are sometimes called *scalars* or *scalar quantities*. To pass an array element to a method, use the subscripted name of the array element as an argument in the method call.

For a method to receive an array through a method call, the method's parameter list must specify that an array will be received. For example, the method header for method `ModifyArray` might be written as

```
public void ModifyArray( int[] b )
```

indicating that `ModifyArray` expects to receive an integer array in parameter `b`. Arrays are passed by reference; when the called method uses the array name `b`, it refers to the actual array in the caller (array `hourlyTemperatures`).

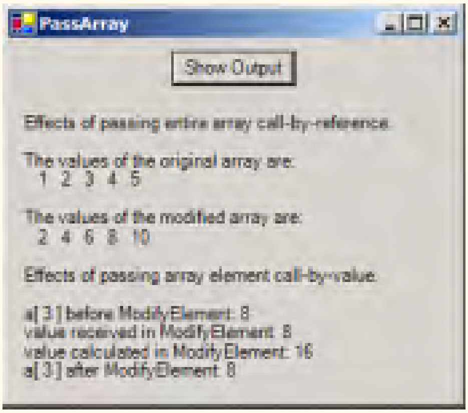
The application in Fig. 7.8 demonstrates the difference between passing an entire array and passing an array element.

The `for` loop on lines 32–33 appends the five elements of integer array `a` to the `Text` property of `outputLabel`. Line 33 invokes method `ModifyArray` and passes to it array `a`. Method `ModifyArray` multiplies each element by 2. To illustrate that array `a`'s elements were modified, the `for` loop on lines 41–42 appends the five elements of integer array `a` to the `Text` property of `outputLabel`. As the screen capture indicates, the elements of `a` are modified by `ModifyArray`.

```
// Fig. 7.8: PassArray.cs
// Passing arrays and individual elements to methods.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class PassArray : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button showOutputButton;
   private System.Windows.Forms.Label outputLabel;

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new PassArray() );
   }

   private void showOutputButton_Click( object sender,
      System.EventArgs e )
   {
      int[] a = { 1, 2, 3, 4, 5 };

      outputLabel.Text = "Effects of passing entire array " +
         "call-by-reference:\n\nThe values of the original " +
         "array are:\n\t";

      for ( int i = 0; i < a.Length; i++ )
         outputLabel.Text += "   " + a[ i ];

      ModifyArray( a );   // array is passed by reference

      outputLabel.Text +=
         "\n\nThe values of the modified array are:\n\t";

      // display elements of array a
      for ( int i = 0; i < a.Length; i++ )
         outputLabel.Text += "   " + a[ i ];

      outputLabel.Text += "\n\nEffects of passing array " +
         "element call-by-value:\n\na[ 3 ] before " +
         "ModifyElement: " + a[ 3 ];

      // array element passed call-by-value
      ModifyElement( a[ 3 ] );

      outputLabel.Text +=
         "\na[ 3 ] after ModifyElement: " + a[ 3 ];
   }
```

Fig. 7.8 Passing arrays and individual array elements to methods. (Part 1 of 2.)

```
54
55       // method modifies the array it receives,
56       // original will be modified
57       public void ModifyArray( int[] b )
58       {
59          for ( int j = 0; j < b.Length; j++ )
60             b[ j ] *= 2;
61       }
62
63       // method modifies the integer passed to it
64       // original will not be modified
65       public void ModifyElement( int e )
66       {
67          outputLabel.Text +=
68             "\nvalue received in ModifyElement: " + e;
69
70          e *= 2;
71
72          outputLabel.Text +=
73             "\nvalue calculated in ModifyElement: " + e;
74       }
75    }
```



```
Effects of passing entire array call-by-reference

The values of the original array are:
   1  2  3  4  5

The values of the modified array are:
   2  4  6  8  10

Effects of passing array element call-by-value

a[ 3 ] before ModifyElement: 8
value received in ModifyElement: 8
value calculated in ModifyElement: 16
a[ 3 ] after ModifyElement: 8
```

Fig. 7.8 Passing arrays and individual array elements to methods. (Part 2 of 2.)

To show the value of **`a[ 3 ]`** before the call to **`ModifyElement`**, lines 44–46 append the value of **`a[ 3 ]`** (and other information) to **`outputLabel.Text`**. Line 44 invokes method **`ModifyElement`** and passes **`a[ 3 ]`**. Remember that **`a[ 3 ]`** is a single **`int`** value in the array **`a`**. Also, remember that values of primitive types always are passed to methods by value. Therefore, a copy of **`a[ 3 ]`** is passed. Method **`ModifyElement`** multiplies its argument by 2 and stores the result in its parameter **`e`**. The parameter of **`ModifyElement`** is a local variable, so when the method terminates, the local variable is destroyed. Thus, when control is returned to **`PassArray`**, the unmodified value of **`a[ 3 ]`** is appended to the **`outputLabel.Text`** (line 51–52).

7.6 Passing Arrays by Value and by Reference

In C#, a variable that “stores” an object, such as an array, does not actually store the object itself. Instead, such a variable stores a reference to the object (i.e., the location in the computer’s memory where the object itself is stored). The distinction between reference variables and primitive data type variables raises some subtle issues that programmers must understand to create secure, stable programs.

When a program passes an argument to a method, the called method receives a copy of that argument’s value. Changes to the local copy do not affect the original variable that the program passed to the method. If the argument is of a reference type, the method makes a local copy of the reference itself, not a copy of the actual object to which the reference refers. The local copy of the reference also refers to the original object in memory. Thus, reference types are always passed by reference, which means that changes to those objects in called methods affect the original objects in memory.

Performance Tip 7.1

Passing arrays and other objects by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would waste time and would consume considerable storage for the copies of the arrays—both of these problems cause poor performance.

C# also allows methods to pass references with keyword `ref`. This is a subtle capability, which, if misused, can lead to problems. For instance, when a reference-type object like an array is passed with `ref`, the called method actually gains control over the passed reference itself, allowing the called method to replace the original reference in the caller with a different object or even with `null`. Such behavior can lead to unpredictable effects, which can be disastrous in mission-critical applications. The program in Fig. 7.9 demonstrates the subtle difference between passing a reference by value and passing a reference with keyword `ref`.

Lines 26 and 29 declare two integer array variables, `firstArray` and `firstArrayCopy` (we make the copy so we can determine whether reference `firstArray` gets overwritten). Line 26 initializes `firstArray` with the values `1`, `2` and `3`. The assignment statement on line 29 copies reference `firstArray` to variable `firstArrayCopy`, causing these variables to reference the same array object in memory. The `for` structure on lines 38–39 prints the contents of `firstArray` before it is passed to method `FirstDouble` (line 42) so we can verify that this array is passed by reference (i.e., the called method indeed changes the array’s contents).

The `for` structure in method `FirstDouble` (lines 99–100) multiplies the values of all the elements in the array by `2`. Line 103 allocates a new array containing the values `11`, `12` and `13`; the reference for this array then is assigned to parameter `array` (in an attempt to overwrite reference `firstArray`—this, of course, will not happen, because the reference was passed by value). After method `FirstDouble` executes, the `for` structure on lines 48–49 prints the contents of `firstArray`, demonstrating that the values of the elements have been changed by the method (and confirming that in C# arrays are always passed by reference). The `if`/`else` structure on lines 52–57 uses the `==` operator to compare references `firstArray` (which we just attempted to overwrite) and `firstArrayCopy`. The expression on line 40 evaluates to `true` if the operands to binary operator `==` indeed reference the same object. In this case, the object represented is the array allocated in line 26—not the array allocated in method `FirstDouble` (line 103).

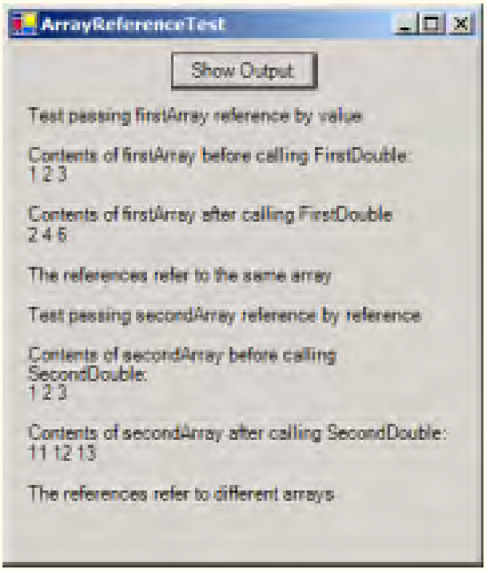
```
// Fig. 7.9: ArrayReferenceTest.cs
// Testing the effects of passing array references
// by value and by reference.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class ArrayReferenceTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label outputLabel;
   private System.Windows.Forms.Button showOutputButton;

   [STAThread]
   static void Main()
   {
      Application.Run( new ArrayReferenceTest() );
   }

   private void showOutputButton_Click( object sender,
      System.EventArgs e )
   {
      // create and initialize firstArray
      int[] firstArray = { 1, 2, 3 };

      // copy firstArray reference
      int[] firstArrayCopy = firstArray;

      outputLabel.Text =
         "Test passing firstArray reference by value";

      outputLabel.Text += "\n\nContents of firstArray " +
         "before calling FirstDouble:\n\t";

      // print contents of firstArray
      for ( int i = 0; i < firstArray.Length; i++ )
         outputLabel.Text += firstArray[ i ] + " ";

      // pass reference firstArray by value to FirstDouble
      FirstDouble( firstArray );

      outputLabel.Text += "\n\nContents of firstArray after " +
         "calling FirstDouble\n\t";

      // print contents of firstArray
      for ( int i = 0; i < firstArray.Length; i++ )
         outputLabel.Text += firstArray[ i ] + " ";

```

Fig. 7.9 Passing an array reference by value and by reference (Part 1 of 3.).

```
      // test whether reference was changed by FirstDouble
      if ( firstArray == firstArrayCopy )
         outputLabel.Text +=
            "\n\nThe references refer to the same array\n";
      else
         outputLabel.Text +=
            "\n\nThe references refer to different arrays\n";

      // create and initialize secondArray
      int[] secondArray = { 1, 2, 3 };

      // copy secondArray reference
      int[] secondArrayCopy = secondArray;

      outputLabel.Text += "\nTest passing secondArray " +
         "reference by reference";

      outputLabel.Text += "\n\nContents of secondArray " +
         "before calling SecondDouble:\n\t";

      // print contents of secondArray before method call
      for ( int i = 0; i < secondArray.Length; i++ )
         outputLabel.Text += secondArray[ i ] + " ";

      SecondDouble( ref secondArray );

      outputLabel.Text += "\n\nContents of secondArray " +
         "after calling SecondDouble:\n\t";

      // print contents of secondArray after method call
      for ( int i = 0; i < secondArray.Length; i++ )
         outputLabel.Text += secondArray[ i ] + " ";

      // test whether reference was changed by SecondDouble
      if ( secondArray == secondArrayCopy )
         outputLabel.Text +=
            "\n\nThe references refer to the same array\n";
      else
         outputLabel.Text +=
            "\n\nThe references refer to different arrays\n";

   } // end method showOutputButton_Click

   // modify elements of array and attempt to modify
   // reference
   void FirstDouble( int[] array )
   {
      // double each element's value
      for ( int i = 0; i < array.Length; i++ )
         array[ i ] *= 2;

```

Fig. 7.9 Passing an array reference by value and by reference (Part 2 of 3.).

```
102         // create new reference and assign it to array
103         array = new int[] { 11, 12, 13 };
104      }
105
106      // modify elements of array and change reference array
107      // to refer to a new array
108      void SecondDouble( ref int[] array )
109      {
110         // double each element's value
111         for ( int i = 0; i < array.Length; i++ )
112            array[ i ] *= 2;
113
114         // create new reference and assign it to array
115         array = new int[] { 11, 12, 13 };
116      }
117   }
```



Test passing firstArray reference by value

Contents of firstArray before calling FirstDouble:
1 2 3

Contents of firstArray after calling FirstDouble
2 4 6

The references refer to the same array

Test passing secondArray reference by reference

Contents of secondArray before calling
SecondDouble:
1 2 3

Contents of secondArray after calling SecondDouble:
11 12 13

The references refer to different arrays

Fig. 7.9 Passing an array reference by value and by reference (Part 3 of 3.).

Lines 60–90 perform similar tests, using array variables **`secondArray`** and **`secondArrayCopy`** and method **`SecondDouble`** (lines 108–116). Method **`SecondDouble`** performs the same operations as **`FirstDouble`**, but receives its array argument using keyword **`ref`**. In this case, the reference stored in **`secondArray`** after the method call is a reference to the array allocated on line 115 of **`SecondDouble`**, demonstrating that a reference passed with keyword **`ref`** can be modified by the called method so that the reference actually points to a different object, in this case an array allocated in procedure **`SecondDouble`**. The **`if`**/**`else`** structure in lines 85–90 demonstrates that **`secondArray`** and **`secondArrayCopy`** no longer refer to the same array.

Software Engineering Observation 7.1

When a method receives a reference-type object parameter by value, the object is not passed by value—the object still passes by reference. Rather, the object's reference is passed by value. This prevents a method from overwriting references passed to that method. In the vast majority of cases, protecting the caller's reference from modification is the desired behavior. If you encounter a situation where you truly want the called procedure to modify the caller's reference, pass the reference-type using keyword **`ref`**—*but, again, such situations are rare.*

Software Engineering Observation 7.2

In C#, reference-type objects (including arrays) always pass by reference. So, a called procedure receiving a reference to an object in a caller can change the caller's object.

7.7 Sorting Arrays

Sorting data (i.e., arranging the data into some particular order, such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name, and within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and in many cases, massive amounts of it. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the computer science field. In this section, we discuss one of the simplest sorting schemes. In the exercises, we investigate more sophisticated sorting algorithms.

Performance Tip 7.2

Sometimes, the simplest algorithms perform poorly. Their virtue is that they are easy to write, test and debug. Complex algorithms sometimes are needed to realize maximum performance of a program.

Figure 7.10 sorts the values of the 10-element array **`a`** into ascending order. The technique we use is called the *bubble sort*, because smaller values gradually "bubble" their way to the top of the array (i.e., toward the first element) like air bubbles rising in water. The technique sometimes is called the *sinking sort*, because the larger values sink to the bottom of the array. Bubble sort uses nested loops to make several passes through the array. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the values remain in the same order. If a pair is in decreasing order, the bubble sort swaps the values in the array. The program contains methods **`Main`**, **`BubbleSort`** and **`Swap`**. Method **`sortButton_Click`** (lines 23–41) creates array **`a`**, invokes **`BubbleSort`** and displays output. Line 34 of **`sortButton_Click`** invokes method **`BubbleSort`** (lines 44–52) to sort array **`a`**. Line 51 in method **`BubbleSort`** calls method **`Swap`** (lines 55–62) to exchange two elements of the array.

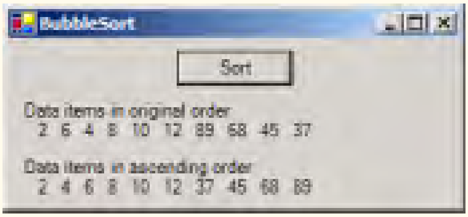
```
1  // Fig. 7.10: BubbleSorter.cs
2  // Sorting an array's values into ascending order.
3  using System;
4  using System.Drawing;
5  using System.Collections;
```

Fig. 7.10 Sorting an array with bubble sort. (Part 1 of 3.)

```
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class BubbleSorter : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button sortButton;
   private System.Windows.Forms.Label outputLabel;

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new BubbleSorter() );
   }

   private void sortButton_Click( object sender,
         System.EventArgs e )
   {
      int[] a =  { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

      outputLabel.Text = "Data items in original order\n";

      for ( int i = 0; i < a.Length; i++ )
         outputLabel.Text += "   " + a[ i ];

      // sort elements in array a
      BubbleSort( a );

      outputLabel.Text += "\n\nData items in ascending order\n";

      for ( int i = 0; i < a.Length; i++ )
         outputLabel.Text += "   " + a[ i ];

   } // end method sortButton_Click

   // sort the elements of an array with bubble sort
   public void BubbleSort( int[] b )
   {
      for ( int pass = 1; pass < b.Length; pass++ ) // passes

         for ( int i = 0; i < b.Length - 1; i++ )   // one pass

            if ( b[ i ] > b[ i + 1 ] )      // one comparison
               Swap( b, i );                // one swap
   }

   // swap two elements of an array
   public void Swap( int[] c, int first )
   {
      int hold;      // temporary holding area for swap

```

Fig. 7.10 Sorting an array with bubble sort. (Part 2 of 3.)

```
59         hold = c[ first ];
60         c[ first ] = c[ first + 1 ];
61         c[ first + 1 ] = hold;
62      }
63   }
```



Data items in original order
2 6 4 8 10 12 89 68 45 37

Data items in ascending order
2 4 6 8 10 12 37 45 68 89

Fig. 7.10 Sorting an array with bubble sort. (Part 3 of 3.)

Method **`BubbleSort`** receives the array as parameter **`b`**. The nested **`for`** loop on lines 46–51 performs the sort. The outer loop controls the number of passes of the array. The inner loop controls the comparisons and necessary swapping of the elements during each pass.

Method **`BubbleSort`** first compares **`b[ 0 ]`** to **`b[ 1 ]`**, then **`b[ 1 ]`** to **`b[ 2 ]`**, then **`b[ 2 ]`** to **`b[ 3 ]`** and so on, until it completes the pass by comparing **`b[ 8 ]`** to **`b[ 9 ]`**. Although there are 10 elements, the comparison loop performs only nine comparisons. As a result of the way the successive comparisons are made, a large value may move down the array (sink) many positions (and sometimes all the way to the bottom of the array) on a single pass. However, a small value may move up (bubble) only one position. On the first pass, the largest value is guaranteed to sink to the bottom element of the array, **`b[ 9 ]`**. On the second pass, the second largest value is guaranteed to sink to **`b[ 8 ]`**. On the ninth pass, the ninth largest value sinks to **`b[ 1 ]`**. This leaves the smallest value in **`b[ 0 ]`**, so only nine passes are needed to sort a 10-element array.

If a comparison reveals that the two elements appear in descending order, **`BubbleSort`** calls **`Swap`** to exchange the two elements so they will be in ascending order in the array. Method **`Swap`** receives a reference to the array (which it calls **`c`**) and one integer representing the subscript of the first element of the array to be exchanged. Three assignments on lines 59–61 perform the exchange, where the extra variable **`hold`** temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```
c[ first ] = c[ first + 1 ];
c[ first + 1 ] = c[ first ];
```

If **`c[ first ]`** is **`7`** and **`c[ first + 1 ]`** is **`5`**, after the first assignment, both elements of the array contain **`5`** and the value **`7`** is lost—hence, the need for the extra variable **`hold`**.

The advantage of the bubble sort is that it is easy to program. However, the bubble sort runs slowly, which becomes apparent when sorting large arrays. More advanced courses (often titled "Data Structures" or "Algorithms" or "Computational Complexity") investigate sorting and searching in greater depth. Note that the .NET framework includes a built-in array-sorting capability that implements a high-speed sort. To sort the array **`a`** in Fig. 7.10, you can use the statement

```
Array.Sort( a );
```

7.8 Searching Arrays: Linear Search and Binary Search

Often, programmers work with large amounts of data stored in arrays. It might be necessary in this case to determine whether an array contains a value that matches a certain *key value*. The process of locating a particular element value in an array is called *searching*. In this section, we discuss two searching techniques—the simple *linear search* technique and the more efficient *binary search* technique. Exercises 7.8 and 7.9 at the end of this chapter ask you to implement recursive versions of the linear and binary search.

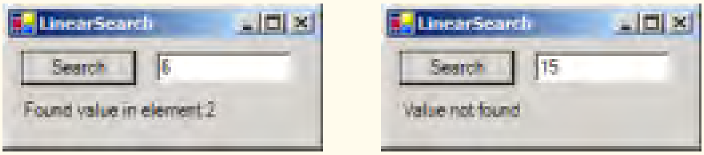
7.8.1 Searching an Array with Linear Search

In the program in Fig. 7.11, method **`LinearSearch`** (defined on lines 44–54) uses a **`for`** structure containing an **`if`** structure to compare each element of an array with a *search key* (line 44). If the search key is found, the method returns the subscript value for the element to indicate the exact position of the search key in the array. If the search key is not found, the method returns **`-1`**. (The value **`-1`** is a good choice because it is not a valid subscript number.) If the elements of the array being searched are not in any particular order, it is just as likely that the value will be found in the first element as in the last. On average, the program will have to compare the search key with half the elements of the array. The program contains a 100-element array filled with the even integers from 0–198. The user types the search key in a **`TextBox`** (called ***`inputTextBox`***) and clicks the **`findButton`** to start the search. [*Note*: The array is passed to **`LinearSearch`** even though the array is an instance variable of the class. This is done because an array normally is passed to a method of another class for searching.]

```
 1  // Fig. 7.11: LinearSearcher.cs
 2  // Demonstrating linear searching of an array.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.Data;
 9
10  public class LinearSearcher : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.Button searchButton;
13     private System.Windows.Forms.TextBox inputTextBox;
14     private System.Windows.Forms.Label outputLabel;
15
16     int[] a = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26,
17                 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50 };
18
19     // Visual Studio .NET generated code
20
21     [STAThread]
22     static void Main()
23     {
```

Fig. 7.11 Linear search of an array. (Part 1 of 2.)

```
      Application.Run( new LinearSearcher() );
   }

   private void searchButton_Click( object sender,
      System.EventArgs e )
   {
      int searchKey = Int32.Parse( inputTextBox.Text );

      int elementIndex = LinearSearch( a, searchKey );

      if ( elementIndex != -1 )
         outputLabel.Text =
            "Found value in element " + elementIndex;

      else
         outputLabel.Text = "Value not found";

   } // end method searchButton_Click

   // search array for the specified key value
   public int LinearSearch( int[] array, int key )
   {
      for ( int n = 0; n < array.Length; n++ )
      {
         if ( array[ n ] == key )
            return n;
      }

      return -1;

   } // end method LinearSearch
}
```



6

Found value in element 2

15

Value not found

Fig. 7.11 Linear search of an array. (Part 2 of 2.)

7.8.2 Searching a Sorted Array with Binary Search

The linear search method works well for small or unsorted arrays. However, for large arrays, linear searching is inefficient. If the array is sorted, the high-speed *binary search* technique can be used. The binary search algorithm eliminates half of the elements in the array being searched after each comparison. The algorithm locates the middle array element and compares it with the search key. If they are equal, the search key has been found, and the subscript of that element is returned. Otherwise, the problem is reduced to searching half of the array. If the search key is less than the middle array element, the first half of the array is searched; otherwise, the second half of the array is searched. If the search key is not the middle element in the specified subarray (a piece of the original array), the algorithm is repeated in one quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).

In a worst-case scenario, searching an array of 1024 elements will take only 10 comparisons by using a binary search. Repeatedly dividing 1024 by 2 (after each comparison we eliminate from consideration half the array) yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1024 (2^{10}) is divided by 2 only ten times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 (2^{20}) elements takes a maximum of 20 comparisons to find the key. An array of one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous increase in performance over the linear search, which required comparing the search key with an average of half the elements in the array. For a one-billion-element array, the difference is between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.

Figure 7.12 presents the iterative version of method **`BinarySearch`** (lines 59–85). The method receives two arguments—an integer array called **`array`** (the array to search) and an integer **`key`** (the search key). The array is passed to **`BinarySearch`** even though the array is an instance variable of the class. Once again, this is done because an array normally is passed to a method of another class for searching. Line 67 calculates the middle element of the array being searched by determining the number of elements in the array and dividing this value by 2. Recall that using the **`/`** operator with integers performs an integer division, which truncates the result. So, when there is an even number of elements in the array there is no "middle" element—the middle of our array is actually between two elements. When this occurs, the calculation on line 67 returns the smaller index of the two middle elements.

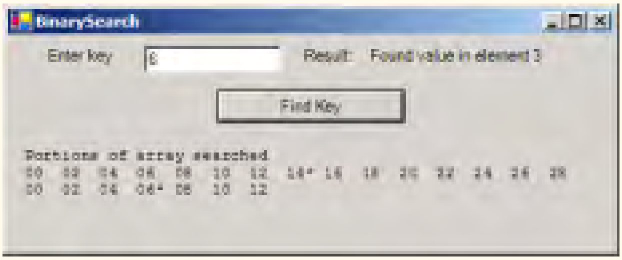
```
 1  // Fig. 7.12: BinarySearchTest.cs
 2  // Demonstrating a binary search of an array.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  public class BinarySearchTest : System.Windows.Forms.Form
12  {
13     private System.Windows.Forms.Label promptLabel;
14
15     private System.Windows.Forms.TextBox inputTextBox;
16
17     private System.Windows.Forms.Label resultLabel;
18     private System.Windows.Forms.Label displayLabel;
19     private System.Windows.Forms.Label outputLabel;
20
21     private System.Windows.Forms.Button findButton;
22
```

Fig. 7.12 Binary search of a sorted array. (Part 1 of 4.)

```
      private System.ComponentModel.Container components = null;

      int[] a = { 0, 2, 4, 6, 8, 10, 12, 14, 16,
                  18, 20, 22, 24, 26, 28 };

      // Visual Studio .NET generated code

      // main entry point for application
      [STAThread]
      static void Main()
      {
         Application.Run( new BinarySearchTest() );
      }

      // searches for an element by calling
      // BinarySearch and displaying results
      private void findButton_Click( object sender,
         System.EventArgs e )
      {
         int searchKey = Int32.Parse( inputTextBox.Text );

         // initialize display string for the new search
         outputLabel.Text = "Portions of array searched\n";

         // perform the binary search
         int element = BinarySearch( a, searchKey );

         if ( element != -1 )
            displayLabel.Text = "Found value in element " +
               element;
         else
            displayLabel.Text = "Value not found";

      } // end findButton_Click

      // searchs array for specified key
      public int BinarySearch( int[] array, int key )
      {
         int low = 0;                       // low subscript
         int high = array.Length - 1;       // high subscript
         int middle;                        // middle subscript

         while ( low <= high )
         {
            middle = ( low + high ) / 2;

            // the following line displays the portion
            // of the array currently being manipulated during
            // each iteration of the binary search loop
            BuildOutput( a, low, middle, high );

            if ( key == array[ middle ] )   // match
               return middle;
```

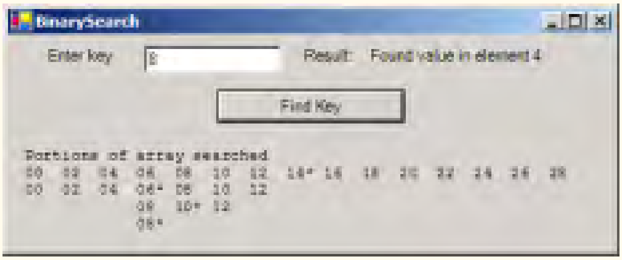
Fig. 7.12 Binary search of a sorted array. (Part 2 of 4.)

```
         else if ( key < array[ middle ] )
            high = middle - 1;   // search low end of array
         else
            low = middle + 1;

      } // end BinarySearch

      return -1;  // search key not found

   } // end method BinarySearch

   public void BuildOutput(
      int[] array, int low, int mid, int high )
   {
      for ( int i = 0; i < array.Length; i++ )
      {
         if ( i < low || i > high )
            outputLabel.Text += "    ";

         // mark middle element in output
         else if ( i == mid )
            outputLabel.Text +=
               array[ i ].ToString( "00" ) + "* ";
         else
            outputLabel.Text +=
               array[ i ].ToString( "00" ) + "  ";
      }

      outputLabel.Text += "\n";

   } // end BuildOutput

} // end class BinarySearchTest
```



Result: Found value in element 3

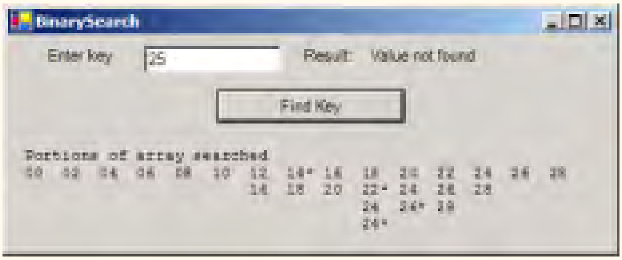
```
Portions of array searched
00  02  04  06  08  10  12  14* 16  18  20  22  24  26  28
00  02  04  06* 08  10  12
```



Result: Found value in element 4

```
Portions of array searched
00  02  04  06  08  10  12  14* 16  18  20  22  24  26  28
00  02  04  06* 08  10  12
            08  10* 12
            08*
```

Fig. 7.12 Binary search of a sorted array. (Part 3 of 4.)



```
Portions of array searched
00  02  04  06  08  10  12  14* 16  18  20  22  24  26  28
                        16  18  20  22* 24  26  28
                                    24  26* 28
                                    24*
```

Fig. 7.12 Binary search of a sorted array. (Part 4 of 4.)

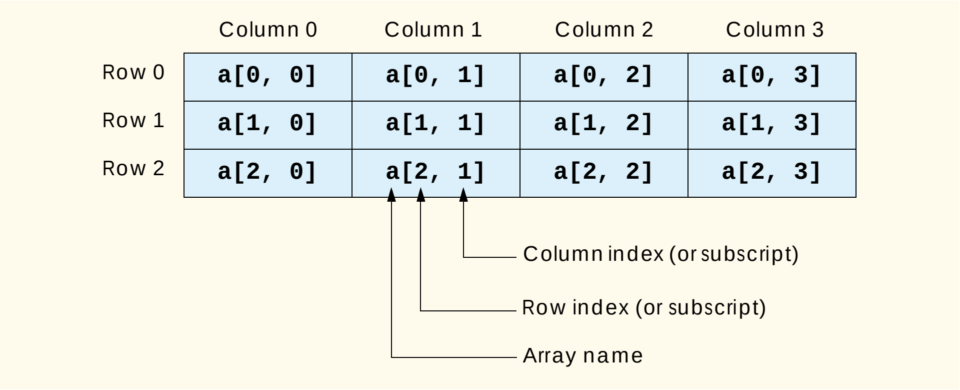
If **`key`** matches the **`middle`** element of a subarray (line 74), **`BinarySearch`** returns **`middle`** (the subscript of the current element), indicating that the value was found and the search is complete. If **`key`** does not match the **`middle`** element of a subarray, **`BinarySearch`** adjusts the **`low`** subscript or **`high`** subscript (both declared in the method) so that a smaller subarray can be searched. If **`key`** is less than the middle element (line 76), the **`high`** subscript is set to **`middle - 1`**, and the search continues on the elements from **`low`** to **`middle - 1`**. If **`key`** is greater than the middle element (line 78), the **`low`** subscript is set to **`middle + 1`**, and the search continues on the elements from **`middle + 1`** to **`high`**. These comparisons occur in the nested **`if`**/**`else`** structure on lines 74–79.

The program uses a 15-element array. The first power of 2 greater than the number of array elements is 16 (2^4)—so at most four comparisons are required to find the **`key`**. To illustrate this concept, method **`BinarySearch`** calls method **`BuildOutput`** (lines 87–106) to output each subarray during the binary search process. **`BuildOutput`** marks the middle element in each subarray with an asterisk (**`*`**) to indicate the element with which the **`key`** is compared. Each search in this example results in a maximum of four lines of output—one per comparison. Note that the .NET framework includes a built-in array-searching capability that implements the binary-search algorithm. To search for the key 7 in the sorted array **`a`** in Fig. 7.12, you can use the statement

```
Array.BinarySearch( a, 7 );
```

7.9 Multiple-Subscripted Arrays

So far we have studied *single-subscripted* (or *one-dimensional*) arrays—i.e., those that contain single lists of values. In this section, we introduce *multiple-subscripted* (often called *multidimensional*) arrays. Such arrays require two or more subscripts to identify particular elements. Arrays that require two subscripts to identify a particular element commonly are called *double-subscripted arrays.* We concentrate on *double-subscripted arrays* (often called *two-dimensional arrays*). There are two types of multiple-subscripted arrays—*rectangular* and *jagged*. Rectangular arrays with two subscripts often represent *tables* of values consisting of information arranged in *rows* and *columns*, where each row is the same size, and each column is the same size. To identify a particular table element, we must specify the two subscripts—by convention, the first identifies the element's row and the second identifies the element's column. Multiple-subscripted arrays can have more than two subscripts. Figure 7.13 illustrates a double-subscripted array, **`a`**, containing three rows and four columns (i.e., a 3-by-4 array). An array with *m* rows and *n* columns is called an *m-by-n array.*



	Column 0	Column 1	Column 2	Column 3
Row 0	`a[0, 0]`	`a[0, 1]`	`a[0, 2]`	`a[0, 3]`
Row 1	`a[1, 0]`	`a[1, 1]`	`a[1, 2]`	`a[1, 3]`
Row 2	`a[2, 0]`	`a[2, 1]`	`a[2, 2]`	`a[2, 3]`

Fig. 7.13 Double-subscripted array with three rows and four columns.

Every element in array **`a`** is identified in Fig. 7.13 by an element name of the form **`a[ i , j ]`**, in which **`a`** is the name of the array, and **`i`** and **`j`** are the subscripts that uniquely identify the row and column of each element in **`a`**. Notice that the names of the elements in the first row all have a first subscript of **`0`**; the names of the elements in the fourth column all have a second subscript of **`3`**.

Multiple-subscripted arrays can be initialized in declarations like single-subscripted arrays. A double-subscripted array **`b`** with two rows and two columns could be declared and initialized with

```
int[,] b = new int[ 2, 2 ];

b[ 0, 0 ] = 1;
b[ 0, 1 ] = 2;
b[ 1, 0 ] = 3;
b[ 1, 1 ] = 4;
```

or this can be written on one line using an *initializer list* as shown below:

```
int[,] b = { { 1, 2 }, { 3, 4 } };
```

The values are grouped by row in braces. Thus, **`1`** and **`2`** initialize **`b[ 0 , 0 ]`** and **`b[ 0 , 1 ]`**, and **`3`** and **`4`** initialize **`b[ 1 , 0 ]`** and **`b[ 1 , 1 ]`**. The compiler determines the number of rows by counting the number of sub-initializer lists (represented by sets of braces) in the main initializer list. The compiler determines the number of columns in each row by counting the number of initializer values in the sub-initializer list for that row. Method **`GetLength`** returns the length of a particular array dimension. In the preceding example, **`b.GetLength( 0 )`** returns the length of the zeroth dimension of **`b`**, which is **`2`**.

Jagged arrays are maintained as arrays of arrays. Unlike in rectangular arrays, the arrays that compose jagged arrays can be of different lengths. The declaration

```
int[][] c = new int[ 2 ][];  // allocate rows

// allocate and initialize elements in row 0
c[ 0 ] = new int[] { 1, 2 };
```

```
// allocate and initialize elements in row 0
c[ 1 ] = new int[] { 3, 4, 5 };
```

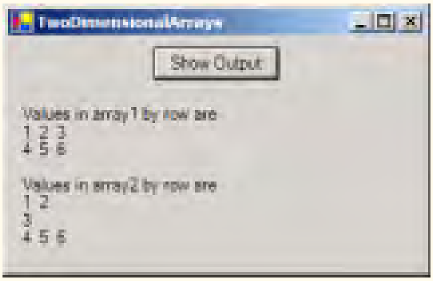
creates integer array **`c`** with row **`0`** (which is an array itself) containing two elements (**`1`** and **`2`**), and row **`1`** containing three elements (**`3`**, **`4`** and **`5`**). The **`Length`** property of each subarray can be used to determine the size of each column. For the jagged array **`c`**, the size of the **`zeroth`** column is **`c[ 0 ].Length`**, which is **`2`**.

The application in Fig. 7.14 demonstrates the initialization of double-subscripted arrays in declarations and the use of nested **`for`** loops to traverse the arrays (i.e., to manipulate each array element).

```
// Fig. 7.14: TwoDimensionalArrays.cs
// Initializing two-dimensional arrays.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class TwoDimensionalArrays : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button showOutputButton;
   private System.Windows.Forms.Label outputLabel;

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new TwoDimensionalArrays() );
   }

   private void showOutputButton_Click( object sender,
      System.EventArgs e )
   {
      // declaration and initialization of rectangular array
      int[,] array1 = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };

      // declaration and initialization of jagged array
      int[][] array2 = new int[ 3 ][];
      array2[ 0 ] = new int[] { 1, 2 };
      array2[ 1 ] = new int[] { 3 };
      array2[ 2 ] = new int[] { 4, 5, 6 };

      outputLabel.Text = "Values in array1 by row are\n";

      // output values in array1
      for ( int i = 0; i < array1.GetLength( 0 ); i++ )
      {
         for ( int j = 0; j < array1.GetLength( 1 ); j++ )
            outputLabel.Text += array1[ i, j ] + "  ";
```

Fig. 7.14 Initializing multidimensional arrays. (Part 1 of 2.)

```
42
43          outputLabel.Text += "\n";
44       }
45
46       outputLabel.Text += "\nValues in array2 by row are\n";
47
48       // output values in array2
49       for ( int i = 0; i < array2.Length; i++ )
50       {
51          for ( int j = 0; j < array2[ i ].Length; j++ )
52             outputLabel.Text += array2[ i ][ j ] + "  ";
53
54          outputLabel.Text += "\n";
55       }
56
57    } // end method showOutputButton_Click
58 }
```



```
Values in array1 by row are
1 2 3
4 5 6

Values in array2 by row are
1 2
3
4 5 6
```

Fig. 7.14 Initializing multidimensional arrays. (Part 2 of 2.)

The declaration of **`array1`** (line 27) provides six initializers in two sublists. The first sublist initializes the first row of the array to the values 1, 2 and 3. The second sublist initializes the second row of the array to the values 4, 5 and 6. The declaration of **`array2`** (line 30) creates a jagged array of 3 arrays (specified by the 3 in the first set of square brackets). Lines 31–33 initialize each subarray so that the first subarray contains the values 1 and 2, the second contains the value 3 and the last contains the values 4, 5 and 6.

The **`for`** structure on lines 38–44 appends the elements of **`array1`** to **`string output`**. Note the use of a nested **`for`** structure to output the rows of each double-subscripted array. In the nested **`for`** structures for **`array1`**, we use method **`GetLength`** to determine the number of elements in each dimension of the array. Line 38 determines the number of rows in the array by invoking **`array1.GetLength( 0 )`**, and line 40 determines the number of columns in the array by invoking **`array1.GetLength( 1 )`**. Arrays with additional dimensions would require more deeply nested **`for`** loops to process.

The nested **`for`** structures on lines 49–55 output the elements of jagged array **`array2`**. Recall that a jagged array is essentially an array that contains additional arrays as its elements. Line 49 uses the **`Length`** property of **`array2`** to determine the number of rows in the jagged array. Line 51 determines the **`Length`** of each subarray with the expression **`array2[ i ].Length`**.

Many common array manipulations use **`for`** repetition structures. For the remainder of this section, we will focus on manipulations of jagged arrays. Imagine a jagged array **`a`**, which contains 3 rows, or arrays. The following **`for`** structure sets all the elements in the third row of array **`a`** to zero:

```
for ( int col = 0; col < a[ 2 ].Length; col++ )
   a[ 2 ][ col ] = 0;
```

We specified the *third* row; therefore, we know that the first subscript is always **`2`** (**`0`** is the first row and **`1`** is the second row). The **`for`** loop varies only the second subscript (i.e., the column subscript). Notice the use of **`a[ 2 ].Length`** in the **`for`** structure's conditional expression. This statement demonstrates that each row of **`a`** is an array in itself, and therefore the program can access a typical array's properties, such as **`Length`**. Assuming the length of array **`a[ 2 ]`** is **`4`**, the preceding **`for`** structure is equivalent to the assignment statements

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

The following nested **`for`** structure determines the total of all the elements in array **`a`**. We use **`a.Length`** in the conditional expression of the outer **`for`** structure to determine the number of rows in **`a`**, in this case, 3.

```
int total = 0;

for ( int row = 0; row < a.Length; row++ )

   for ( int col = 0; col < a[ row ].Length; col++ )
      total += a[ row ][ col ];
```

The **`for`** structure totals the elements of the array one row at a time. The outer **`for`** structure begins by setting the **`row`** subscript to **`0`**, so the elements of the first row may be totaled by the inner **`for`** structure. Then the outer **`for`** structure increments **`row`** to **`1`**, so the second row can be totaled. Finally, the outer **`for`** structure increments **`row`** to **`2`**, so the third row can be totaled. The result can be displayed when the nested **`for`** structure terminates.

The program in Fig. 7.15 performs several other array manipulations on 3-by-4 array **`grades`**. Each row of the array represents a student, and each column represents a grade on one of the four exams that the student took during the semester. The array manipulations are performed by four methods. Method **`Minimum`** (lines 64–76) determines the lowest grade of any student for the semester. Method **`Maximum`** (lines 79–91) determines the highest grade of any student for the semester. Method **`Average`** (lines 94–102) determines a particular student's semester average.

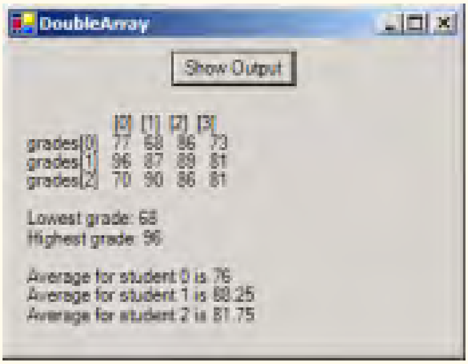
Methods **`Minimum`** and **`Maximum`** use array **`grades`** and the variables **`students`** (number of rows in the array) and **`exams`** (number of columns in the array). Each method loops through array **`grades`** by using nested **`for`** structures. Consider the nested **`for`** structure from method **`Minimum`** (lines 68–73). The outer **`for`** structure sets **`i`** (i.e., the row subscript) to **`0`** so the elements of the first row can be compared with variable **`lowGrade`** in the body of the inner **`for`** structure. The inner **`for`** structure loops through the four grades of a particular row and compares each grade with **`lowGrade`**. If a grade is less than **`lowGrade`**, then **`lowGrade`** is set to that grade. The outer **`for`** structure then increments the row subscript by **`1`**. The elements of the second row are compared with variable **`lowGrade`**. The outer **`for`** structure then increments the row subscript to **`2`**. The elements

```
// Fig. 7.15: DoubleArray.cs
// Manipulating a double-subscripted array.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class DoubleArray : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button showOutputButton;
   private System.Windows.Forms.Label outputLabel;

   int[][] grades;
   int students, exams;

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new DoubleArray() );
   }

   private void showOutputButton_Click( object sender,
      System.EventArgs e )

   {
      grades = new int[ 3 ][];
      grades[ 0 ] =  new int[]{ 77, 68, 86, 73 };
      grades[ 1 ] =  new int[]{ 96, 87, 89, 81 };
      grades[ 2 ] =  new int[]{ 70, 90, 86, 81 };

      students = grades.Length;      // number of students
      exams = grades[ 0 ].Length;    // number of exams

      // line up column headings
      outputLabel.Text = "                   ";

      // output the column headings
      for ( int i = 0; i < exams; i++ )
         outputLabel.Text += "[" + i + "]  ";

      // output the rows
      for ( int i = 0; i < students; i++ )
      {
         outputLabel.Text += "\ngrades[" + i + "]   ";

         for ( int j = 0; j < exams; j++ )
            outputLabel.Text += grades[ i ][ j ] + "   ";
      }
```

Fig. 7.15 Example using double-subscripted arrays. (Part 1 of 3.)

```

      outputLabel.Text += "\n\nLowest grade: " + Minimum() +
         "\nHighest grade: " + Maximum() + "\n";

      for ( int i = 0; i < students; i++ )
        outputLabel.Text += "\nAverage for student " + i + " is " +
            Average( grades[ i ] );

   } // end method showOutputButton_Click

   // find minimum grade in grades array
   public int Minimum()
   {
      int lowGrade = 100;

      for ( int i = 0; i < students; i++ )

         for ( int j = 0; j < exams; j++ )

            if ( grades[ i ][ j ] < lowGrade )
               lowGrade = grades[ i ][ j ];

      return lowGrade;
   }

   // find maximum grade in grades array
   public int Maximum()
   {
      int highGrade = 0;

      for ( int i = 0; i < students; i++ )

         for ( int j = 0; j < exams; j++ )

            if ( grades[ i ][ j ] > highGrade )
               highGrade = grades[ i ][ j ];

      return highGrade;
   }

   // determine average grade for a particular student
   public double Average( int[] setOfGrades )
   {
      int total = 0;

      for ( int i = 0; i < setOfGrades.Length; i++ )
         total += setOfGrades[ i ];

      return ( double ) total / setOfGrades.Length;
   }

} // end class DoubleArray
```

Fig. 7.15 Example using double-subscripted arrays. (Part 2 of 3.)



```
          [0]  [1]  [2]  [3]
grades[0]  77   68   86   73
grades[1]  96   87   89   81
grades[2]  70   90   86   81

Lowest grade: 68
Highest grade: 96

Average for student 0 is 76
Average for student 1 is 88.25
Average for student 2 is 81.75
```

Fig. 7.15 Example using double-subscripted arrays. (Part 3 of 3.)

of the third row are compared with variable **`lowGrade`**. When execution of the nested structure is complete, **`lowGrade`** contains the smallest grade in the double-subscripted array. Method **`Maximum`** works similarly to method **`Minimum`**.

Method **`Average`** takes one argument—a single-subscripted array of test results for a particular student. When **`Average`** is called (line 59), the argument **`grades[ i ]`** specifies that a particular row of the double-subscripted array **`grades`** is to be passed to **`Average`**. For example, the argument **`grades[ 1 ]`** represents the four values (a single-subscripted array of grades) stored in the second row of the double-subscripted array **`grades`**. Remember that a jagged two-dimensional array is an array with elements that are single-subscripted arrays. Method **`Average`** calculates the sum of the array elements, divides the total by the number of test results and then returns the floating-point result cast as a **`double`** value (line 101).

7.10 `foreach` Repetition Structure

C# provides the **`foreach`** retition structure for iterating through values in data structures, such as arrays. When used with one-dimensional arrays, **`foreach`** behaves like a **`for`** structure that iterates through the range of indices from **`0`** to the array's **`Length`**. Instead of a counter, **`foreach`** uses a variable to represent the value of each element. The program in Fig. 7.16 uses the **`foreach`** structure to determine the minimum value in a two-dimensional array of grades.

```
// Fig. 7.16: ForEach.cs
// Demonstrating for/each structure.
using System;

class ForEach
{
   // main entry point for the application
   static void Main( string[] args )
   {
```

Fig. 7.16 Using **`For Each`**/**`Next`** with an array. (Part 1 of 2.)

```
10          int[,] gradeArray = { { 77, 68, 86, 73 },
11             { 98, 87, 89, 81 }, { 70, 90, 86, 81 } };
12
13          int lowGrade = 100;
14
15          foreach ( int grade in gradeArray )
16          {
17             if ( grade < lowGrade )
18                lowGrade = grade;
19          }
20
21          Console.WriteLine( "The minimum grade is: " + lowGrade );
22       }
23    }
```

```
The minimum grade is: 68
```

Fig. 7.16 Using **For Each/Next** with an array. (Part 2 of 2.)

The header of the **foreach** structure (line 15) specifies a variable, **grade**, and an array, **gradeArray**. The **foreach** structure iterates through all elements in **gradeArray**, sequentially assigning each value to variable **grade**. Line 15 compares each value to variable **lowGrade**, which stores the lowest grade in the array.

For rectangular arrays, the repetition of the **foreach** structure begins with the element whose indices are all zero, then iterates through all possible combinations of indices, incrementing the rightmost index first. When the rightmost index reaches its upper bound, it is reset to zero, and the index to the left of it is incremented by 1. In this case, **grade** takes the values as they are ordered in the initializer list in lines 10–11. When all the grades have been processed, **lowGrade** is displayed (line 21).

Although many array calculations are handled best with a counter, **foreach** is useful when the indices of the elements are not important. The **foreach** structure is particularly useful for looping through arrays of objects, as we discuss in Chapter 10, Object-Oriented Programming: Polymorphism.

SUMMARY

- An array is a group of contiguous memory locations that all have the same name and type.
- To refer to a particular location or element in an array, specify the name of the array and the position number of the element within the array.
- The first element in every array is the zeroth element (i.e., element 0).
- The position number in square brackets is more formally called a subscript (or an index). This number must be an integer or an integer expression.
- To reference the i^{th} element of a single-dimensional array, use i-1 as the index.
- The brackets that enclose the subscript of an array are operators that have the same level of precedence as parentheses.
- When arrays are allocated, the elements are initialized to zero for the numeric primitive-data-type variables, to **false** for **bool** variables or to **null** for reference types.
- Arrays may be declared to contain most data types.

- In an array of primitive data types, every element of the array contains one value of the declared data type of the array.
- In an array of a reference type, every element of the array is a reference to an object of the data type of the array. For example, every element of a **`string`** array is a reference to a **`string`** and that reference has the value **`null`** by default.
- The elements of single-dimensional and rectangular arrays can be allocated and initialized in the array declaration by following the declaration with an equal sign and a comma-separated initializer list enclosed in braces (**`{`** and **`}`**).
- A **`const`** variable must be declared and initialized in the same statement.
- Constants also are called named constants. They often are used to make a program more readable.
- Unlike its predecessors C and C++, .NET-compliant languages provide mechanisms to prevent accessing elements outside the bounds of the array.
- When a reference is made to a nonexistent element of an array, an **`IndexOutOfRangeException`** occurs.
- To pass an array argument to a method, specify the name of the array without any brackets.
- Although entire arrays are passed by reference, individual array elements of primitive data types are passed by value, as are simple variables.
- To pass an array element to a method, use the subscripted name of the array element as an argument in the method call.
- Sorting data (i.e., placing the data into a particular order, such as ascending or descending) is one of the most important computing applications.
- The chief virtue of the bubble sort is that it is easy to program. However, the bubble sort runs slowly, which becomes apparent when sorting large arrays.
- The linear search method works well for small or unsorted arrays. However, for large arrays, linear searching is inefficient.
- After each comparison, the binary search algorithm eliminates from consideration half the elements in the array being searched. The algorithm locates the middle array element and compares it to the search key. If they are equal, the search key has been found, and the subscript of that element is returned. Otherwise, the problem is reduced to searching half the array. If the search key is less than the middle array element, the first half of the array is searched; otherwise, the second half of the array is searched. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).
- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 that is greater than the number of elements in the array.
- There are two types of multiple-subscripted arrays—rectangular and jagged.
- In general, an array with *m* rows and *n* columns is referred to as an *m*-by-*n* array.
- Multiple-subscripted arrays can be initialized in declarations, as can single-subscripted arrays.
- The compiler determines the number of columns in each row by counting the number of initializer values in the sub-initializer list for that row.
- Jagged arrays are maintained as arrays of arrays. Unlike rectangular arrays, rows in jagged arrays can be of different lengths.
- Many common array manipulations use **`for`** repetition structures.
- When used with one-dimensional arrays, **`foreach`** behaves like a **`for`** structure that iterates through the range of indices from **`0`** to the array's **`Length`**.

- For rectangular arrays, the repetition of the **`foreach`** structure begins with the element whose indices are all zero, then iterates through all possible combinations of indices, incrementing the rightmost index first. When the rightmost index reaches its upper bound, it is reset to zero, and the index to the left of it is incremented by 1.

TERMINOLOGY

`[]`, subscript operator
array allocated with **`new`**
array automatically initialized to zeros
array bounds
array declaration
array of arrays (jagged array)
bar chart
binary search algorithm
brute force
bubble sort
column
`const`
constant variable
declare an array
dice-rolling program
double-subscripted array
element
exception
`foreach` structure
graph information
histogram
ignoring element zero
initializer list
initializing double-subscripted arrays in declarations
innermost set of square brackets
invalid array reference
jagged array
key value
length of an array
linear search
lvalue (“left value”)
m-by-*n* array
multiple-subscripted array
named constant
nested **`for`** loop
`new` operator
`null`
“off-by-one error”
one-dimensional array
partition
partitioning step
pass of a bubble sort
passing array to method
passing array element to method
position number
read-only variable
rectangular array
search key
searching
single-subscripted array
sinking sort
size of an array
sorting
square brackets, **`[]`**
subarray
sub-initializer list
subscript
swap
table
table element
`TextBox`
“walk” past end of an array
zero-based counting
zeroth element

SELF-REVIEW EXERCISES

7.1 Fill in the blanks in each of the following statements:
a) Lists and tables of values can be stored in __________.
b) The elements of an array are related by the fact that they have the same __________ and __________.
c) The number that refers to a particular element of an array is called its __________.
d) The process of placing the elements of an array in order is called __________ the array.
e) Determining if an array contains a certain key value is called __________ the array.
f) Arrays that use two or more subscripts are referred to as __________ arrays.

g) __________ arrays are maintained as arrays of arrays.
h) A __________ variable must be declared and initialized in the same statement, or a syntax error will occur.
i) C# provides the __________ repetition structure for iterating through values in data structures, such as arrays.
j) When an invalid array reference is made, an __________ is generated.

7.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) An array can store many different types of values at the same time.
b) An array subscript normally should be of data type **`float`**.
c) An individual array element that is passed to a method and modified in that method will contain the modified value when the called method completes execution.
d) The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array.
e) There are two types of multiple-subscripted arrays—square and jagged.
f) A **`const`** variable must be declared and initialized in the same statement, or a syntax error will occur.
g) After each comparison, the binary search algorithm eliminates from consideration one third of the elements in the portion of the array that is being searched.
h) To determine the number of elements in an array, we can use the **`NumberOfElements`** property.
i) The linear search method works well for small or unsorted arrays.
j) In an *m*-by-*n* array, the *m* stands for the number of columns and the *n* stands for the number of rows.

ANSWERS TO SELF-REVIEW EXERCISES

7.1 a) arrays. b) name, type. c) subscript, index or position number. d) sorting. e) searching. f) multiple-subscripted. g.) Jagged. h) **`const`**. i) **`foreach`**. j) **`IndexOutofRangeException`**.

7.2 a) False. An array can store only values of the same type. b) False. An array subscript must be an integer or an integer expression. c) False. For individual primitive-data-type elements of an array, they are passed by value. If a reference to an array element is passed, then modifications to that array element are reflected in the original. An individual element of a reference type is passed to a method by reference. d) True. e) False. The two different types are called rectangular and jagged. f) True. g) False. After each comparison, the binary search algorithm eliminates from consideration half the elements in the portion of the array that is being searched. h) False. To determine the number of elements in an array, we can use the **`Length`** property. i) True. j) False. In an *m*-by-*n* array, the *m* stands for the number of rows and the *n* stands for the number of columns.

EXERCISES

7.3 Write statements to accomplish each of the following tasks:
a) Display the value of the seventh element of character array **`f`**.
b) Initialize each of the five elements of single-subscripted integer array **`g`** to **`8`**.
c) Total the elements of floating-point array **`c`** of 100 elements.
d) Copy 11-element array **`a`** into the first portion of array **`b`** containing 34 elements.
e) Determine the smallest and largest values contained in 99-element floating-point array **`w`**.

7.4 Use a single-subscripted array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive $200 per week, plus 9% of their gross sales for that week. For example, a salesperson who grosses $5000 in sales in a week receives $200 plus 9% of $5000, or a total of $650. Write a program (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a) \$200–\$299
b) \$300–\$399
c) \$400–\$499
d) \$500–\$599
e) \$600–\$699
f) \$700–\$799
g) \$800–\$899
h) \$900–\$999
i) \$1000 and over

7.5 Use a single-subscripted array to solve the following problem: Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it is not a duplicate of a number already read. Provide for the "worst case" (in which all 20 numbers are different). Use the smallest possible array to solve this problem.

7.6 (*Turtle Graphics*) The Logo language made famous the concept of *turtle graphics*. Imagine a mechanical turtle that walks around the room under the control of a program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about without writing anything. In this problem, you will simulate the operation of the turtle and create a computerized sketchpad.

Use a 20-by-20 array **`floor`**, which is initialized to zeros. Read commands from an array that contains them. At all times, keep track of the current position of the turtle and whether the pen is up or down. Assume that the turtle always starts at position 0,0 of the floor with its pen up. The set of turtle commands your program must process are as follows:

Command	Meaning
`1`	Pen up
`2`	Pen down
`3`	Turn right
`4`	Turn left
`5,10`	Move forward 10 spaces (or a number other than 10)
`6`	Print the 20-by-20 array
`9`	End of data (sentinel)

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square, leaving the pen in the up position:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array `floor` to `1`s. When the `6` command (print) is given, wherever there is a `1` in the array, display an asterisk or another character. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities we have discussed. Write several turtle graphics programs to draw interesting shapes. Add commands to increase the power of your turtle graphics language.

SPECIAL SECTION: RECURSION EXERCISES

7.7 (*Palindromes*) A palindrome is a string that is spelled the same forward and backward. Some examples of palindromes are “radar,” “able was i ere i saw elba” and, if blanks are ignored, “a man a plan a canal panama.” Write a recursive method `testPalindrome` that returns `true` if the string stored in the array is a palindrome and `false` otherwise. The method should ignore spaces and punctuation in the string.

7.8 (*Linear Search*) Modify Fig. 7.11 to use recursive method `LinearSearch` to perform a linear search of the array. The method should receive an integer array and the size of the array as arguments. If the search key is found, return the array subscript; otherwise, return –1.

7.9 (*Binary Search*) Modify the program in Fig. 7.12 to use a recursive method `BinarySearch` to perform the binary search of the array. The method should receive an integer array and the starting and ending subscript as arguments. If the search key is found, return the array subscript; otherwise, return –1.

7.10 (*Quicksort*) In this chapter, we discussed the sorting technique bubble sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

a) *Partitioning Step.* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.
b) *Recursive Step.* Perform step 1 on each unsorted subarray.

Each time Step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, it must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple, but how do we determine the final position of the first element of each subarray? Consider the following set of values (partitioning element in bold—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost element of the array, compare each element to **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The new array is

12 2 6 4 89 8 10 **37** 68 45

Element 12 is italicized to indicate that it was just swapped with **37**.

b) Starting from the left of the array, but beginning with the element after 12, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The new array is

12 2 6 4 **37** 8 10 *89* 68 45

c) Starting from the right, but beginning with the element before 89, compare each element to **37** until an element less than **37** is found, then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The new array is

12 2 6 4 *10* 8 **37** 89 68 45

d) Starting from the left, but beginning with the element after 10, compare each element to **37** until an element greater than **37** is found, then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** to itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the previous array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Using the preceding discussion, write recursive method **`QuickSort`** to sort a single-subscripted integer array. The method should receive as arguments an integer array, a starting subscript and an ending subscript. Method **`Partition`** should be called by **`QuickSort`** to perform the partitioning step.

7.11 (*Maze Traversal*) The following grid of **#**s and dots (**`.`**) is a double-subscripted array representation of a maze:

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

The **#**s represent the walls of the maze, and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

Write recursive method **`MazeTraverse`** to walk through the maze. The method should receive as arguments a 12-by-12 character array representing the maze and the starting location of the maze. As **`MazeTraverse`** attempts to locate the exit from the maze, it should place the character **`X`** in each square in the path. The method should display the maze after each move so the user can watch as the maze is solved.

8

Object-Based Programming

Objectives

- To understand encapsulation and data hiding.
- To understand the concepts of data abstraction and abstract data types (ADTs).
- To be able to create, use and destroy objects.
- To be able to control access to object instance variables and methods.
- To be able to use properties to keep objects in consistent states.
- To understand the use of the `this` reference.
- To understand namespaces and assemblies.
- To be able to use the **Class View** and **Object Browser**.

My object all sublime
I shall achieve in time.
W. S. Gilbert

Is it a world to hide virtues in?
William Shakespeare, *Twelfth Night*

Your public servants serve you right.
Adlai Stevenson

Classes struggle, some classes triumph, others are eliminated.
Mao Zedong

This above all: to thine own self be true.
William Shakespeare, *Hamlet*

Outline

8.1 Introduction
8.2 Implementing a Time Abstract Data Type with a Class
8.3 Class Scope
8.4 Controlling Access to Members
8.5 Initializing Class Objects: Constructors
8.6 Using Overloaded Constructors
8.7 Properties
8.8 Composition: Objects References as Instance Variables of Other Classes
8.9 Using the `this` Reference
8.10 Garbage Collection
8.11 `static` Class Members
8.12 `const` and `readonly` Members
8.13 Indexers
8.14 Data Abstraction and Information Hiding
8.15 Software Reusability
8.16 Namespaces and Assemblies
8.17 Class View and Object Browser

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

8.1 Introduction

In this chapter, we investigate object orientation in C#. Some readers might ask, why have we deferred this topic until now? There are several reasons. First, the objects we build in this chapter partially are composed of structured program pieces. To explain the organization of objects, we needed to establish a basis in structured programming with control structures. We also wanted to study methods in detail before introducing object orientation. Finally, we wanted to familiarize readers with arrays, which are C# objects.

In our discussions of object-oriented programs in Chapters 1–7, we introduced many basic concepts (i.e., “object think”) and terminology (i.e., “object speak”) that relate to C# object-oriented programming. We also discussed our program-development methodology: We analyzed typical problems that required programs to be built and determined what classes from the .NET Framework Class Library were needed to implement each program. We then selected appropriate instance variables and methods for each program and specified the manner in which an object of our class collaborated with objects from the .NET Framework classes to accomplish the program’s overall goals.

Let us briefly review some key concepts and terminology of object orientation. Object orientation uses classes to *encapsulate* (i.e., wrap together) data (*attributes*) and methods (*behaviors*). Objects have the ability to hide their implementation from other objects (this principle is called *information hiding*). Although some objects can communicate with one another across well-defined *interfaces* (just like the driver's interface to a car includes a steering wheel, accelerator pedal, brake pedal and gear shift), objects are unaware of how other objects are implemented (just as the driver is unaware of how the steering, engine, brake and transmission mechanisms are implemented). Normally, implementation details are hidden within the objects themselves. Surely, it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work. Later, we will see why information hiding is so crucial to good software engineering.

In *procedural programming languages* (like C), programming tends to be *action oriented*. C# programming, however, is *object oriented*. In C, the unit of programming is the *function* (functions are called *methods* in C#). In C#, the unit of programming is the *class*. Objects eventually are *instantiated* (i.e., created) from these classes and functions are encapsulated within the "boundaries" of classes as methods.

C programmers concentrate on writing functions. They group actions that perform some task into a function and then group functions to form a program. Data are certainly important in C, but they exist primarily to support the actions that functions perform. The *verbs* in a system-requirements document describing the requirements for a new application help a C programmer determine the set of functions that will work together to implement the system.

By contrast, C# programmers concentrate on creating their own *user-defined types*, called *classes*. We also refer to classes as *programmer-defined types*. Each class contains both data and a set of methods that manipulate the data. The data components, or *data members*, of a class are called *member variables*, or *instance variables* (many C# programmers prefer the term *fields*).[1] Just as we call an instance of a built-in type—such as **`int`**—a *variable*, we call an instance of a user-defined type (i.e., a class) an *object*. In C#, attention is focused on classes, rather than on methods. The *nouns* in a system-requirements document help the C# programmer determine an initial set of classes with which to begin the design process. Programmers use these classes to instantiate objects that work together to implement the system.

This chapter explains how to create and use classes and objects, a subject known as *object-based programming (OBP)*. Chapters 9 and 10 introduce *inheritance* and *polymorphism*—key technologies that enable *object-oriented programming (OOP)*. Although we do not discuss inheritance in detail until Chapter 9, it is part of several class definitions in this chapter and has been used in several examples previously. For example, in the program of Section 4.13 (and several subsequent programs), we inherited a class from **`System.Windows.Forms.Form`** to create an application that executes in its own window.

Software Engineering Observation 8.1

All C# objects are passed by reference.

8.2 Implementing a Time Abstract Data Type with a Class

Classes in C# facilitate the creation of *abstract data types (ADT)*, which hide their implementation from clients (or users of the class object). A problem in procedural programming languages is that client code often is dependent on implementation details of the data used in the code. This dependency might necessitate rewriting the client code if the data implementation changes. ADTs eliminate this problem by providing implementation-independent *interfaces* to their clients. The creator of a class can change the internal implementation of that class without affecting the clients of that class.

---

[1] We sometimes use industry-standard terminology, such as data members and instance members. rather than C# terms such as fields, For a listing of C#-specific terminology, please see the C# Language Specification, which can be downloaded from **`msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp`**.

Software Engineering Observation 8.2

It is important to write programs that are understandable and easy to maintain. Change is the rule, rather than the exception. Programmers should anticipate that their code will be modified. As we will see, classes facilitate program modifiability.

The following example (an susequent examples) will require multiple class definitions in the same project. To add a class to a project, select **Add Class…** from the **Project** menu. In the **Add New Item** dialog box that appears, enter the new class name in the **Name** text box and click the **Open** button. Note that the file name (ending with the **`.cs`** file extension) appears in the **Solution Explorer** below the project name.

Our next example consists of classes **`Time1`** (Fig. 8.1) and **`TimeTest1`** (Fig. 8.2). Class **`Time1`** contains the time of day in 24-hour clock format. Class **`TimeTest1`** contains method **`Main`**, which creates an instance of class **`Time1`** and demonstrates the features of that class.

```
// Fig. 8.1: Time1.cs
// Class Time1 maintains time in 24-hour format.

using System;

// Time1 class definition
public class Time1 : Object
{
   private int hour;    // 0-23
   private int minute;  // 0-59
   private int second;  // 0-59

   // Time1 constructor initializes instance variables to
   // zero to set default time to midnight
   public Time1()
   {
      SetTime( 0, 0, 0 );
   }

   // Set new time value in 24-hour format. Perform validity
   // checks on the data. Set invalid values to zero.
   public void SetTime(
      int hourValue, int minuteValue, int secondValue )
   {
      hour = ( hourValue >= 0 && hourValue < 24 ) ?
         hourValue : 0;
      minute = ( minuteValue >= 0 && minuteValue < 60 ) ?
         minuteValue : 0;
```

Fig. 8.1 **`Time1`** abstract data type represents the time in 24-hour format. (Part 1 of 2.)

```
29          second = ( secondValue >= 0 && secondValue < 60 ) ?
30             secondValue : 0;
31 
32       } // end method SetTime
33 
34       // convert time to universal-time (24 hour) format string
35       public string ToUniversalString()
36       {
37          return String.Format(
38             "{0:D2}:{1:D2}:{2:D2}", hour, minute, second );
39       }
40 
41       // convert time to standard-time (12 hour) format string
42       public string ToStandardString()
43       {
44          return String.Format( "{0}:{1:D2}:{2:D2} {3}",
45             ( ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ),
46             minute, second, ( hour < 12 ? "AM" : "PM" ) );
47       }
48 
49    } // end class Time1
```

Fig. 8.1 `Time1` abstract data type represents the time in 24-hour format. (Part 2 of 2.)

In Fig. 8.1, line 7 begins the **`Time1`** class definition, indicating that class **`Time1`** inherits from class ***`Object`*** (namespace **`System`**). C# programmers use *inheritance* to create classes from existing classes. In fact, every class in C# (except **`Object`**) inherits from an existing class definition. On line 7, the **`:`** followed by class name **`Object`** indicates that class **`Time1`** inherits existing pieces of class **`Object`**. If a new class definition does not specify a **`:`** and class name to the right of the new class name, the new class implicitly inherits from class **`Object`**. It is not necessary to understand inheritance to learn the concepts and programs in this chapter. We explore inheritance and class **`Object`** in detail in Chapter 9.

The opening left brace (**`{`**) at line 8 and closing right brace (**`}`**) at line 49 delineate the *body* of class **`Time1`**. Any information that we place in this body is said to be encapsulated (i.e., wrapped) in the class. For example, lines 9–11 of class **`Time1`** declare three **`int`** variables—**`hour`**, **`minute`** and **`second`**—that represent the time of day in *universal-time* format (*24-hour clock* format). Variables declared in a class definition, but not inside a method definition, are called *instance variables*—each instance (object) of the class contains its own separate copy of the class's instance variables.

Keywords ***`public`*** and ***`private`*** are *member access modifiers*. Instance variables or methods with member access modifier **`public`** are accessible wherever the program has a reference to a **`Time1`** object. However, instance variables or methods declared with member access modifier **`private`** are accessible only in that class definition. A class's **`public`** members and **`private`** members can be intermixed.

Good Programming Practice 8.1

*Every instance variable or method definition should be preceded by a member access modifier. The default access modifier for class members is **`private`**.*

Good Programming Practice 8.2

Members in a class definition should be grouped by their member access modifiers to enhance clarity and readability.

Lines 9–11 declare each of the three `int` instance variables—`hour`, `minute` and `second`—with member access modifier `private`, indicating that these instance variables of the class are accessible only to members of the class—this is known as *data hiding*. When an object of the class encapsulates such instance variables, only methods of that object’s class can access the variables. Normally, instance variables are declared `private` and methods are declared `public`. However, it is possible to have `private` methods and `public` instance variables, as we will see later. Often, `private` methods are called *utility methods*, or *helper methods*, because they can be called only by other methods of that class. The purpose of utility methods is to support the operation of a class’s other methods. Using `public` data in a class is an uncommon and dangerous programming practice. Providing such access to data members is unsafe—foreign code (i.e., code in other classes) could set `public` data members to invalid values, producing potentially disastrous results.

Good Programming Practice 8.3

We prefer to list instance variables of a class first, so that, when reading the code, programmers see the name and type of each instance variable before it is used in the methods of the class.

Good Programming Practice 8.4

Even though `private` *and* `public` *members can be intermixed, list all the* `private` *members of a class first in one group, then list all the* `public` *members in another group.*

Software Engineering Observation 8.3

Declare all instance variables of a class as `private`*. The architecture of accessing* `private` *data through* `public` *properties which first validate the data allows the developer to ensure that an object’s data remains in a consistent state.*

Software Engineering Observation 8.4

Make a class member `private` *if there is no reason for that member to be accessed outside of the class definition.*

Classes often include *access methods* that can read or display data. Another common use for access methods is to test the truth of conditions—such methods often are called *predicate methods*. For example, we could design predicate method `IsEmpty` for a *container class*—a class capable of holding many objects, such as a linked list, a stack or a queue. (These data structures are discussed in detail in Chapter 23, Data Structures.) `IsEmpty` would return `true` if the container is empty and `false` otherwise. A program might test `IsEmpty` before attempting to read another item from the container object. Similarly, a program might test another predicate method (e.g., `IsFull`) before attempting to insert an item into a container object.

Class `Time1` contains constructor `Time1` (lines 15–18) and methods `SetTime` (lines 22–32), `ToUniversalString` (lines 35–39) and `ToStandardString` (lines 42–47). These are the *`public` methods* (also called the *`public` services* or the *`public` interface*) of the class. *Clients* of class `Time1`, such as class `TimeTest1` (Fig. 8.2), use a `Time1`’s `public` methods to manipulate the data stored in `Time1` objects or to cause class `Time1` to perform some service.

Lines 15–18 define the *constructor* of class `Time1`. A class's constructor initializes objects of that class. When a program creates an object of class `Time1` with operator `new`, the constructor automatically is called to initialize the object. Class `Time1`'s constructor calls method `SetTime` (lines 22–32) to initialize instance variables `hour`, `minute` and `second` to `0` (representing midnight). Constructors can take arguments, but cannot return values. As we will see, a class can have overloaded constructors. An important difference between constructors and other methods is that constructors cannot specify a return type. Generally, constructors are declared `public`. Note that the constructor name must be the same as the class name.

Common Programming Error 8.1

Attempting to `return` *a value from a constructor is a syntax error.*

Method `SetTime` (lines 22–32) is a `public` method that receives three `int` parameters and uses them to set the time. A conditional expression tests each argument to determine whether the value is in a specified range. For example, the `hour` value must be greater than or equal to 0 and less than 24, because universal-time format represents hours as integers from 0 to 23. Similarly, both minute and second values must be greater than or equal to 0 and less than 60. Any values outside these ranges are invalid values and default to zero. Setting invalid values to zero ensures that a `Time1` object always contains valid data (because, in this example, zero is a valid value for `hour`, `minute` and `second`). When users supply invalid data to `SetTime`, the program might want to indicate that the time was invalid. In Chapter 11, we discuss exception handling, which can be used to indicate invalid initialization values.

Software Engineering Observation 8.5

Always define a class so that each of its instance variables always contains valid values.

Method `ToUniversalString` (lines 35–39) takes no arguments and returns a `string` in universal-time format, consisting of six digits—two for the hour, two for the minute and two for the second. For example, if the time were 1:30:07 PM, method `ToUniversalString` would return `13:30:07`. Lines 37–38 use `String` method `Format` to configure the universal time string. Line 37 passes to `Format` the *format string* `"{0:D2}:{1:D2}:{2:D2}"`, which contains several *format specifications* indicating that arguments `0`, `1` and `2` (the first three arguments after the format string argument) should each have the format `D2` (a two-digit base 10 decimal number format) for display purposes. The `D2` format specification causes single-digit values to appear as two digits with a leading `0` (e.g., `8` would be represented as `08`). The two colons that separate the curly braces `}` and `{` are the colons that separate the hour from the minute and the minute from the second in the resulting `string`.

Method `ToStandardString` (lines 42–47) takes no arguments and returns a `string` in standard-time format, consisting of the `hour`, `minute` and `second` values separated by colons and followed by an AM or a PM indicator (e.g., `1:27:06 PM`). Like method `ToUniversalString`, method `ToStandardString` uses `String` method `Format` to format the `minute` and `second` as two-digit values with leading zeros if necessary. Line 45 determines the value for `hour` in the `string`—if the `hour` is `0` or `12` (AM or PM), the `hour` appears as 12; otherwise, the `hour` appears as a value from 1–11.

After defining the class, we can use it as a type in declarations such as

```
Time1 sunset;  // reference to a Time1 object
```

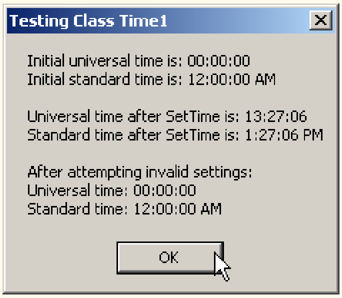
The class name (**`Time1`**) is a type name. A class can yield many objects, just as a primitive data type, such as **`int`**, can yield many variables. Programmers can create class types as needed; this is one reason why C# is known as an *extensible language*.

Class **`TimeTest1`** (Fig. 8.2) uses an instance of class **`Time1`**. Method **`Main`** (lines 11–40) declares and initializes **`Time1`** instance **`time`** (line 13). When the object is instantiated, *operator* ***`new`*** allocates the memory in which the **`Time1`** object will be stored, then calls the **`Time1`** constructor (lines 15–18 of Fig. 8.1) to initialize the instance variables of the **`Time1`** object. As mentioned before, this constructor invokes method **`SetTime`** of class **`Time1`** to initialize each **`private`** instance variable to **`0`**. Operator **`new`** (line 13 of Fig. 8.2) then returns a reference to the newly created object; this reference is assigned to **`time`**.

```
 1  // Fig. 8.2: TimeTest1.cs
 2  // Demonstrating class Time1.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // TimeTest1 uses creates and uses a Time1 object
 8  class TimeTest1
 9  {
10     // main entry point for application
11     static void Main( string[] args )
12     {
13        Time1 time = new Time1();  // calls Time1 constructor
14        string output;
15
16        // assign string representation of time to output
17        output = "Initial universal time is: " +
18           time.ToUniversalString() +
19           "\nInitial standard time is: " +
20           time.ToStandardString();
21
22        // attempt valid time settings
23        time.SetTime( 13, 27, 6 );
24
25        // append new string representations of time to output
26        output += "\n\nUniversal time after SetTime is: " +
27           time.ToUniversalString() +
28           "\nStandard time after SetTime is: " +
29           time.ToStandardString();
30
31        // attempt invalid time settings
32        time.SetTime( 99, 99, 99 );
33
```

Fig. 8.2 Using an abstract data type. (Part 1 of 2.)

```
34       output += "\n\nAfter attempting invalid settings: " +
35          "\nUniversal time: " + time.ToUniversalString() +
36          "\nStandard time: " + time.ToStandardString();
37
38       MessageBox.Show( output, "Testing Class Time1" );
39
40    } // end method Main
41
42 } // end class TimeTest1
```



```
Initial universal time is: 00:00:00
Initial standard time is: 12:00:00 AM

Universal time after SetTime is: 13:27:06
Standard time after SetTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Fig. 8.2 Using an abstract data type. (Part 2 of 2.)

Software Engineering Observation 8.6

Note the relationship between operator **`new`** *and the constructor of a class. When operator* **`new`** *creates an object of a class, that class's constructor is called to initialize the object's instance variables.*

Note that the **`TimeTest.cs`** file does not use keyword **`using`** to import the namespace that contains class **`Time1`**. If a class is in the same namespace as the class that uses it, the **`using`** statement is not required. Every class in C# is part of a namespace. If a programmer does not specify a namespace for a class, the class is placed in the *default namespace*, which includes all compiled classes in the current directory that do not reside in a namespace. In Visual Studio, this current directory is the one in which the current project resides. We must specify **`using`** statements for classes from the .NET Framework, because they are defined outside the namespace of each new application we create. Note that **`using`** statements are not required if the program fully qualifies the name of each class by preceding the class name with its namespace name and a dot operator. For example, a program can invoke class **`MessageBox`**'s **`Show`** method as follows:

```
System.Windows.Forms.MessageBox.Show( "Your message here" );
```

However, such lengthy names can be cumbersome.

Line 14 declares **`string`** reference **`output`** to store the **`string`** containing the results, which later will be displayed in a **`MessageBox`**. Lines 17–20 assign to **`output`** the time in universal-time format (by invoking method **`ToUniversalString`** of the **`Time1`** object) and standard-time format (by invoking method **`ToStandardString`** of the **`Time1`** object). Note the syntax of the method call in each case—the reference **`time`** is followed by a the member access operator (**`.`**) followed by the method name. The reference name specifies the object that will receive the method call.

Line 23 sets the time for the `Time1` object to which `time` refers by passing valid hour, minute and second arguments to `Time1` method `SetTime`. Lines 26–29 append to `output` the new time in both universal and standard formats to confirm that the time was set correctly.

To illustrate that method `SetTime` validates the values passed to it, line 32 passes invalid time arguments to method `SetTime`. Lines 34–36 append to `output` the new time in both formats. All three values passed to `SetTime` are invalid, so instance variables `hour`, `minute` and `second` are set to `0`. Line 38 displays a `MessageBox` with the results of our program. Notice in the last two lines of the output window that the time was indeed set to midnight when invalid arguments were passed to `SetTime`.

`Time1` is our first example of a class that does not contain method `Main`. Thus, class `Time1` cannot be used to begin program execution. Class `TimeTest1` defines a `Main` method, so class `TimeTest1` can be used to begin program execution. A class containing method `Main` also is known as the *entry point* into the program.

Note that the program declares instance variables `hour`, `minute` and `second` as `private`. Such instance variables are not accessible outside the class in which they are defined. A class's clients should not be concerned with the data representation of that class. Clients of a class should be interested only in the services provided by that class. For example, the class could represent the time internally as the number of seconds that have elapsed since the previous midnight. Suppose the data representation changes. Clients still are able to use the same `public` methods and obtain the same results without being aware of the change in internal representation. In this sense, the implementation of a class is said to be *hidden* from its clients.

Software Engineering Observation 8.7

Information hiding promotes program modifiability and simplifies the client's perception of a class.

Software Engineering Observation 8.8

Clients of a class can (and should) use the class without knowing the internal details of how the class is implemented. If the class implementation changes (to improve performance, for example), but the class interface remains constant, the client's source code need not change. This makes it much easier to modify systems.

In this program, the `Time1` constructor initializes the instance variables to `0` (the universal time equivalent of 12 AM) to ensure that the object is created in a *consistent state*—i.e., all instance variables have valid values. The instance variables of a `Time1` object cannot store invalid values, because the constructor, which calls `SetTime`, is called to initialize the instance variables when the `Time1` object is created. Method `SetTime` scrutinizes subsequent attempts by a client to modify the instance variables.

Normally, the instance variables of a class are initialized in that class's constructor, but they also can be initialized when they are declared in the class body. If a programmer does not initialize instance variables explicitly, the compiler implicitly initializes them. When this occurs, the compiler sets primitive numeric variables to `0`, `bool` values to `false` and references to `null`.

Methods `ToUniversalString` and `ToStandardString` take no arguments, because, by default, these methods manipulate the instance variables of the particular `Time1` object on which they are invoked. This often makes method calls more concise than conventional function calls in procedural programming langauages. It also reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

Software Engineering Observation 8.9

The use of an object-oriented programming approach often simplifies method calls by reducing the number of parameters that must be passed. This benefit of object-oriented programming derives from the fact that encapsulation of instance variables and methods within an object gives the object’s methods the right to access the object’s instance variables.

Classes simplify programming, because the client need be concerned only with the **`public`** operations encapsulated in the object. Usually, such operations are designed to be client-oriented, rather than implementation-oriented. Clients are neither aware of, nor involved in, a class’s implementation. Interfaces change less frequently than do implementations. When an implementation changes, implementation-dependent code must change accordingly. By hiding the implementation, we eliminate the possibility that other program parts will become dependent on the class-implementation details.

Often, programmers do not have to create classes “from scratch.” Rather, they can derive classes from other classes that provide behaviors required by the new classes. Classes also can include references to objects of other classes as members. Such *software reuse* can greatly enhance programmer productivity. Chapter 9 discusses *inheritance*—the process by which new classes are derived from existing classes. Section 8.8 discusses *composition* (or *aggregation*), in which classes include as members references to objects of other classes.

8.3 Class Scope

In Section 6.13, we discussed method scope; now, we discuss class *scope*. A class’s instance variables and methods belong to that class’s scope. Within a class’s scope, class members are immediately accessible to all of that class’s methods and can be referenced by name. Outside a class’s scope, class members cannot be referenced directly by name. Those class members that are visible (such as **`public`** members) can be accessed only through a “handle” (i.e., members can be referenced via the format *referenceName*`.`*memberName*).

If a variable is defined in a method, only that method can access the variable (i.e., the variable is a local variable of that method). Such variables are said to have *block scope*. If a method defines a variable that has the same name as a variable with class scope (i.e., an instance variable), the method-scope variable hides the class-scope variable in that method’s scope. A hidden instance variable can be accessed in a method by preceding its name with the keyword ***`this`*** and the dot operator, as in **`this.hour`**. We discuss keyword **`this`** in Section 8.9.

8.4 Controlling Access to Members

The member access modifiers **`public`** and **`private`** control access to a class’s data and methods. (In Chapter 9, we introduce the additional access modifiers **`protected`** and **`internal`**.)

As previously stated, **`public`** methods present to the class's clients a view of the *services* that the class provides (i.e., the **`public`** interface of the class). Previously, we mentioned the merits of writing methods that perform only one task. If a method must execute other tasks to calculate its final result, these tasks should be performed by a helper method. A client does not need to call these helper methods, nor does it need to be concerned with how the class uses its helper methods. For these reasons, helper methods are declared as **`private`** members of a class.

Common Programming Error 8.2

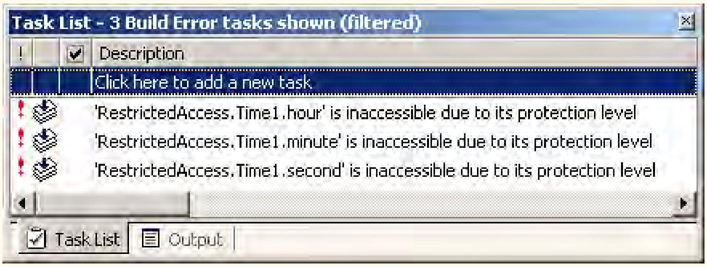
Attempting to access a **`private`** *class member from outside that class is a compiler error.*

The application of Fig. 8.3 (which uses the **`Time1`** class from Fig. 8.1) demonstrates that **`private`** class members are not accessible outside the class. Lines 12–14 attempt to access the **`private`** instance variables **`hour`**, **`minute`** and **`second`** of the **`Time1`** object to which **`time`** refers. When this program is compiled, the compiler generates errors stating that the **`private`** members **`hour`**, **`minute`** and **`second`** are not accessible.

Access to **`private`** data should be controlled carefully by a class's methods. To allow clients to read the values of **`private`** data, the class can define a *property* that enables client code to access this **`private`** data safely. Properties, which we discuss in detail in Section 8.7, contain *accessor methods* that handle the details of modifying and returning data. A property definition can contain a ***`get`*** *accessor*, a ***`set`*** *accessor* or both. A **`get`** accessor enables a client to read a **`private`** data value; a **`set`** accessor enables the client to modify that value. Such modification would seem to violate the notion of **`private`** data. However, a **`set`** accessor can provide data-validation capabilities (such as range checking) to ensure that the value is set properly. A **`set`** accessor also can translate between the format of the data used in the interface and the format used in the underlying implementation. Similarly, a **`get`** accessor need not expose the data in "raw" format; rather, the **`get`** accessor can edit the data and limit the client's view of that data.

```
 1  // Fig. 8.3: RestrictedAccess.cs
 2  // Demonstrate compiler errors from attempt to access
 3  // private class members.
 4
 5  class RestrictedAccess
 6  {
 7     // main entry point for application
 8     static void Main( string[] args )
 9     {
10        Time1 time = new Time1();
11
12        time.hour = 7;
13        time.minute = 15;
14        time.second = 30;
15     }
16
17  } // end class RestrictedAccess
```

Fig. 8.3 Accessing **`private`** class members from client code generates syntax errors. (Part 1 of 2.)


Task List - 3 Build Error tasks shown (filtered)
!
☑
Description
Click here to add a new task
Task List
Output

'RestrictedAccess.Time1.hour' is inaccessible due to its protection level
'RestrictedAccess.Time1.minute' is inaccessible due to its protection level
'RestrictedAccess.Time1.second' is inaccessible due to its protection level

Fig. 8.3 Accessing **`private`** class members from client code generates syntax errors. (Part 2 of 2.)

Software Engineering Observation 8.10

Class designers need not provide **`set`** *or* **`get`** *accessors for each* **`private`** *data member; these capabilities should be provided only when doing so makes sense.*

Software Engineering Observation 8.11

Declaring the instance variables of a class as **`private`** *and the methods and properties of the class as* **`public`** *facilitates debugging, because problems with data manipulations are localized to the class methods that manipulate that data.*

8.5 Initializing Class Objects: Constructors

When a program creates an instance of a class, the program invokes the class's constructor to initialize the class's instance variables (data members). A class can contain overloaded constructors to provide multiple ways to initialize objects of that class. Instance variables can be initialized either by a constructor or when they are declared in the class body. Regardless of whether instance variables receive explicit initialization values, the instance variables always are initialized. In such cases, instance variables receive their default values (`0` for primitive numeric type variables, **`false`** for **`bool`** variable and **`null`** for references).

Performance Tip 8.1

Because instance variables always are initialized to default values by the runtime, avoid initializing instance variables to their default values in the constructor.

Software Engineering Observation 8.12

When appropriate, provide a constructor to ensure that every object is initialized with meaningful values.

When creating an object of a class, the programmer can provide *initializers* in parentheses to the right of the class name. These initializers are the arguments to the constructor. In general, declarations take the form:

ClassName objectReference `=` **`new`** *ClassName* `(` *arguments* `);`

where *objectReference* is a reference of the appropriate data type, **`new`** indicates that an object is being created, *ClassName* indicates the type of the new object (and the name of the constructor being called) and *arguments* specifies a comma-separated list of the values used by the constructor to initialize the object. Figure 8.4 demonstrates using initializers and overloaded constructors.

If a class does not define any constructors, the compiler provides a *default* (*no-argument*) *constructor*. This compiler-provided default constructor contains no code (i.e., the constructor has an empty body) and takes no parameters. The programmer also can provide a default constructor, as we demonstrated in class **`Time1`** (Fig. 8.1). Programmer-provided default constructors can have code in their bodies.

Common Programming Error 8.3

If a class has constructors, but none of the **`public`** *constructors is a default constructor, and a program attempts to call a no-argument constructor to initialize an object of the class, a compilation error occurs. A constructor can be called with no arguments only if there are no constructors for the class (in which case the compiler-provided default constructor is called) or if the class defines a* **`public`** *no-argument constructor.*

8.6 Using Overloaded Constructors

Like methods, constructors of a class can be overloaded. The **`Time1`** constructor in Fig. 8.1 initialized **`hour`**, **`minute`** and **`second`** to **`0`** (i.e., 12 midnight in universal time) via a call to the class **`SetTime`** method. However, class **`Time2`** (Fig. 8.4) overloads the constructor to provide a variety of ways to initialize **`Time2`** objects. Each constructor calls **`Time2`** method **`SetTime`**, which ensures that the object begins in a consistent state by setting out-of-range values to zero. C# invokes the appropriate constructor by matching the number, types and order of the arguments specified in the constructor call with the number, types and order of the parameters specified in each constructor definition.

```
// Fig. 8.4: Time2.cs
// Class Time2 provides overloaded constructors.

using System;

// Time2 class definition
public class Time2
{
   private int hour;     // 0-23
   private int minute;   // 0-59
   private int second;   // 0-59

   // Time2 constructor initializes instance variables to
   // zero to set default time to midnight
   public Time2()
   {
      SetTime( 0, 0, 0 );
   }

   // Time2 constructor: hour supplied, minute and second
   // defaulted to 0
   public Time2( int hour )
   {
      SetTime( hour, 0, 0 );
   }
```

Fig. 8.4 Overloaded constructors provide flexible object-initialization options. (Part 1 of 2.)

```
26
27    // Time2 constructor: hour and minute supplied, second
28    // defaulted to 0
29    public Time2( int hour, int minute )
30    {
31       SetTime( hour, minute, 0 );
32    }
33
34    // Time2 constructor: hour, minute and second supplied
35    public Time2( int hour, int minute, int second )
36    {
37       SetTime( hour, minute, second );
38    }
39
40    // Time2 constructor: initialize using another Time2 object
41    public Time2( Time2 time )
42    {
43       SetTime( time.hour, time.minute, time.second );
44    }
45
46    // Set new time value in 24-hour format. Perform validity
47    // checks on the data. Set invalid values to zero.
48    public void SetTime(
49       int hourValue, int minuteValue, int secondValue )
50    {
51       hour = ( hourValue >= 0 && hourValue < 24 ) ?
52          hourValue : 0;
53       minute = ( minuteValue >= 0 && minuteValue < 60 ) ?
54          minuteValue : 0;
55       second = ( secondValue >= 0 && secondValue < 60 ) ?
56          secondValue : 0;
57    }
58
59    // convert time to universal-time (24 hour) format string
60    public string ToUniversalString()
61    {
62       return String.Format(
63          "{0:D2}:{1:D2}:{2:D2}", hour, minute, second );
64    }
65
66    // convert time to standard-time (12 hour) format string
67    public string ToStandardString()
68    {
69       return String.Format( "{0}:{1:D2}:{2:D2} {3}",
70          ( ( hour == 12 || hour == 0 ) ? 12 : hour % 12 ),
71          minute, second, ( hour < 12 ? "AM" : "PM" ) );
72    }
73
74 } // end class Time2
```

Fig. 8.4 Overloaded constructors provide flexible object-initialization options. (Part 2 of 2.)

Because most of the code in class **`Time2`** is identical to that in class **`Time1`**, this discussion concentrates only on the overloaded constructors. Lines 15–18 define the no-argument constructor that sets the time to midnight. Lines 22–25 define a **`Time2`** constructor that receives a single **`int`** argument representing the **`hour`** and sets the time using the specified **`hour`** value and zero for the **`minute`** and **`second`**. Lines 29–32 define a **`Time2`** constructor that receives two **`int`** arguments representing the **`hour`** and **`minute`** and sets the time using those values and zero for the **`second`**. Lines 35–38 define a **`Time2`** constructor that receives three **`int`** arguments representing the **`hour`**, **`minute`** and **`second`** and uses those values to set the time. Lines 41–44 define a **`Time2`** constructor that receives a reference to another **`Time2`** object. When this last constructor is called, the values from the **`Time2`** argument are used to initialize the **`hour`**, **`minute`** and **`second`** values of the new **`Time2`** object. Even though class **`Time2`** declares **`hour`**, **`minute`** and **`second`** as **`private`** (lines 9–11), the **`Time2`** constructor can access these values in its **`Time2`** argument directly using the expressions **`time.hour`**, **`time.minute`** and **`time.second`**.

Software Engineering Observation 8.13

When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods (including those that are **`private`***).*

Notice that the second, third and fourth constructors (lines 22, 29 and 35) have some arguments in common and that those arguments are kept in the same order. For instance, the constructor that begins on line 29 has as its two arguments an integer representing the hour and an integer representing the minute. The constructor on line 35 has these same two arguments in the same order, followed by its last argument (an integer representing the second).

Good Programming Practice 8.5

When defining overloaded constructors, keep the order of arguments as similar as possible; this makes client programming easier.

Constructors do not specify return types; doing so results in syntax errors. Also, notice that each constructor receives a different number or different types of arguments. Even though only two of the constructors receive values for the **`hour`**, **`minute`** and **`second`**, each constructor calls **`SetTime`** with values for **`hour`**, **`minute`** and **`second`** and uses zeros for the missing values to satisfy **`SetTime`**'s requirement of three arguments.

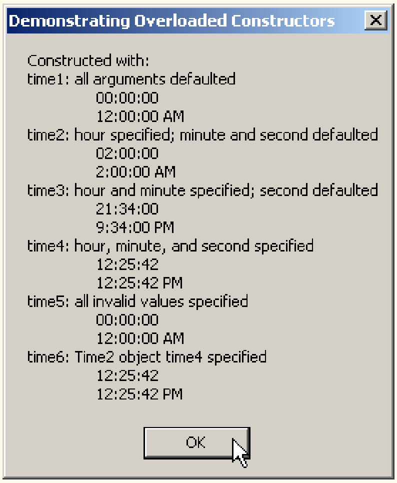
Class **`TimeTest2`** (Fig. 8.5) starts the application that demonstrates the use of overloaded constructors (Fig. 8.4). Lines 15–20 create six **`Time2`** objects that invoke various constructors of the class. Line 15 invokes the no-argument constructor by placing an empty set of parentheses after the class name. Lines 16–20 invoke the **`Time2`** constructors that receive arguments. To invoke the appropriate constructor, pass the proper number, types and order of arguments (specified by the constructor's definition) to that constructor. For example, line 16 invokes the constructor that is defined in lines 22–25 of Fig. 8.4. Lines 22–47 invoke methods **`ToUniversalString`** and **`ToStandardString`** for each **`Time2`** object to demonstrate that the constructors initialize the objects correctly.

```
1  // Fig. 8.5: TimeTest2.cs
2  // Using overloaded constructors.
3
```

Fig. 8.5 Overloaded constructor demonstration. (Part 1 of 3.)

```
using System;
using System.Windows.Forms;

// TimeTest2 demonstrates constructors of class Time2
class TimeTest2
{
   // main entry point for application
   static void Main( string[] args )
   {
      Time2 time1, time2, time3, time4, time5, time6;

      time1 = new Time2();               // 00:00:00
      time2 = new Time2( 2 );            // 02:00:00
      time3 = new Time2( 21, 34 );       // 21:34:00
      time4 = new Time2( 12, 25, 42 );   // 12:25:42
      time5 = new Time2( 27, 74, 99 );   // 00:00:00
      time6 = new Time2( time4 );        // 12:25:42

      String output = "Constructed with: " +
         "\ntime1: all arguments defaulted" +
         "\n\t" + time1.ToUniversalString() +
         "\n\t" + time1.ToStandardString();

      output += "\ntime2: hour specified; minute and " +
         "second defaulted" +
         "\n\t" + time2.ToUniversalString() +
         "\n\t" + time2.ToStandardString();

      output += "\ntime3: hour and minute specified; " +
         "second defaulted" +
         "\n\t" + time3.ToUniversalString() +
         "\n\t" + time3.ToStandardString();

      output += "\ntime4: hour, minute, and second specified" +
         "\n\t" + time4.ToUniversalString() +
         "\n\t" + time4.ToStandardString();

      output += "\ntime5: all invalid values specified" +
         "\n\t" + time5.ToUniversalString() +
         "\n\t" + time5.ToStandardString();

      output += "\ntime6: Time2 object time4 specified" +
         "\n\t" + time6.ToUniversalString() +
         "\n\t" + time6.ToStandardString();

      MessageBox.Show( output,
         "Demonstrating Overloaded Constructors" );

   } // end method Main

} // end class TimeTest2
```

Fig. 8.5 Overloaded constructor demonstration. (Part 2 of 3.)



```
Constructed with:
time1: all arguments defaulted
          00:00:00
          12:00:00 AM
time2: hour specified; minute and second defaulted
          02:00:00
          2:00:00 AM
time3: hour and minute specified; second defaulted
          21:34:00
          9:34:00 PM
time4: hour, minute, and second specified
          12:25:42
          12:25:42 PM
time5: all invalid values specified
          00:00:00
          12:00:00 AM
time6: Time2 object time4 specified
          12:25:42
          12:25:42 PM
```

Fig. 8.5 Overloaded constructor demonstration. (Part 3 of 3.)

Each **`Time2`** constructor can be written to include a copy of the appropriate statements from method **`SetTime`**. This might be slightly more efficient, because it eliminates the extra call to **`SetTime`**. However, consider what would happen if the programmer were to change the representation of the time from three **`int`** values (requiring 12 bytes of memory) to a single **`int`** value representing the total number of seconds that have elapsed in the day (requiring 4 bytes of memory). Placing identical code in the **`Time2`** constructors and method **`SetTime`** makes such a change in the class definition more difficult, because every constructor's body would require modifications to manipulate the data as a single **`int`** rather than three **`int`**s. If the **`Time2`** constructors call **`SetTime`** directly, any changes to the implementation of **`SetTime`** must be made only once, in the body of **`SetTime`**. This reduces the likelihood of introducing a programming error when altering the implementation, because we make only one change in the class, rather than changing every constructor and method **`SetTime`**.

Software Engineering Observation 8.14

If a method of a class provides functionality required by a constructor (or other method) of the class, call that method from the constructor (or other method). This simplifies the maintenance of the code and reduces the likelihood of introducing errors into the code.

8.7 Properties

Methods of a class can manipulate that class's **`private`** instance variables. A typical manipulation might be the adjustment of a customer's bank balance—a **`private`** instance variable of a class **`BankAccount`**—by a **`ComputeInterest`** method.

Classes often provide **`public`** *properties* to allow clients to *set* (i.e., assign values to) or *get* (i.e., obtain the values of) **`private`** instance variables. For example, in Fig. 8.6, we create three properties—**`Hour`**, **`Minute`** and **`Second`**—which access variables **`hour`**, **`minute`** and **`second`**, respectively. Each property contains a ***`get`*** *accessor* (to retrieve the field value) and a ***`set`*** *accessor* (to modify the field value).

Providing **`set`** and **`get`** capabilities appears to be the same as making the instance variables **`public`**. However, this is another one of C#'s subtleties that makes the language so attractive from a software-engineering standpoint. If an instance variable is **`public`**, the instance variable can be read or written to by any method in the program. If an instance variable is **`private`**, a **`public get`** accessor seems to allow other methods to read the data at will. However, the **`get`** accessor can control the formatting and display of the data. Similarly, a **`public set`** accessor can scrutinize attempts to modify the instance variable's value, thus ensuring that the new value is appropriate for that data member. For example, an attempt to **`set`** the day of the month to 37 would be rejected, and an attempt to **`set`** a person's weight to a negative value would be rejected. So, **`set`** and **`get`** accessors can provide access to **`private`** data, but the implementation of these accessors controls what the client code can do to the data.

The declaration of instance variables as **`private`** does not guarantee their integrity. Programmers must provide validity checking—C# provides only the framework with which programmers can design better programs.

Testing and Debugging Tip 8.1

Methods that set the values of **`private`** *data should verify that the intended new values are valid; if they are not, the* **`set`** *accessors should place the* **`private`** *instance variables into an appropriate consistent state.*

The **`set`** accessors of a property cannot return values indicating a failed attempt to assign invalid data to objects of the class. Such return values could be useful to a client of a class when handling errors. The client could take appropriate actions if the objects occupy invalid states. Chapter 11 presents exception handling—a mechanism that can be used to indicate attempts to set an object's members to invalid values.

Figure 8.6 enhances our **`Time`** class, now called **`Time3`**, to include properties for the **`private`** instance variables **`hour`**, **`minute`** and **`second`**. The **`set`** accessors of these properties strictly control the setting of the instance variables to valid values. An attempt to set any instance variable to an incorrect value causes the instance variable to be set to zero (thus leaving the instance variable in a consistent state). Each **`get`** accessor returns the appropriate instance variable's value. This application also introduces enhanced GUI event-handling techniques, as we define a GUI (Fig. 8.7) that includes several buttons the user can click to manipulate the time stored in a **`Time3`** object.

```
// Fig. 8.6: Time3.cs
// Class Time2 provides overloaded constructors.

using System;

// Time3 class definition
public class Time3
{
   private int hour;    // 0-23
   private int minute;  // 0-59
   private int second;  // 0-59

```

Fig. 8.6 Properties provide controlled access to an object's data. (Part 1 of 3.)

```
      // Time3 constructor initializes instance variables to
      // zero to set default time to midnight
      public Time3()
      {
         SetTime( 0, 0, 0 );
      }

      // Time3 constructor: hour supplied, minute and second
      // defaulted to 0
      public Time3( int hour )
      {
         SetTime( hour, 0, 0 );
      }

      // Time3 constructor: hour and minute supplied, second
      // defaulted to 0
      public Time3( int hour, int minute )
      {
         SetTime( hour, minute, 0 );
      }

      // Time3 constructor: hour, minute and second supplied
      public Time3( int hour, int minute, int second )
      {
         SetTime( hour, minute, second );
      }

      // Time3 constructor: initialize using another Time3 object
      public Time3( Time3 time )
      {
         SetTime( time.Hour, time.Minute, time.Second );
      }

      // Set new time value in 24-hour format. Perform validity
      // checks on the data. Set invalid values to zero.
      public void SetTime(
         int hourValue, int minuteValue, int secondValue )
      {
         Hour = hourValue;        // invoke Hour property set
         Minute = minuteValue;    // invoke Minute property set
         Second = secondValue;    // invoke Second property set
      }

      // property Hour
      public int Hour
      {
         get
         {
            return hour;
         }

```

Fig. 8.6 Properties provide controlled access to an object's data. (Part 2 of 3.)

```
         set
         {
            hour = ( ( value >= 0 && value < 24 ) ? value : 0 );
         }

      } // end property Hour

      // property Minute
      public int Minute
      {
         get
         {
            return minute;
         }

         set
         {
            minute = ( ( value >= 0 && value < 60 ) ? value : 0 );
         }

      } // end property Minute

      // property Second
      public int Second
      {
         get
         {
            return second;
         }

         set
         {
            second = ( ( value >= 0 && value < 60 ) ? value : 0 );
         }

      } // end property Second

      // convert time to universal-time (24 hour) format string
      public string ToUniversalString()
      {
         return String.Format(
            "{0:D2}:{1:D2}:{2:D2}", Hour, Minute, Second );
      }

      // convert time to standard-time (12 hour) format string
      public string ToStandardString()
      {
         return String.Format( "{0}:{1:D2}:{2:D2} {3}",
            ( ( Hour == 12 || Hour == 0 ) ? 12 : Hour % 12 ),
            Minute, Second, ( Hour < 12 ? "AM" : "PM" ) );
      }

   } // end class Time3
```

Fig. 8.6 Properties provide controlled access to an object's data. (Part 3 of 3.)

Lines 57–69, 72–84 and 87–99 define **`Time3`** properties **`Hour`**, **`Minute`** and **`Second`**, respectively. Each property begins with a declaration line that includes the property's access modifier (**`public`**), type (**`int`**) and name (**`Hour`**, **`Minute`** or **`Second`**).

The body of each property contains **`get`** and **`set`** accessors, which are declared using the reserved words **`get`** and **`set`**. The **`get`** accessor declarations are on lines 59–62, 74–77 and 89–92. These accessors return the **`hour`**, **`minute`** and **`second`** instance variable values that objects request. The **`set`** accessors are declared on lines 64–67, 79–82 and 94–97. The body of each **`set`** accessor performs the same conditional statement that was previously performed by method **`SetTime`** to set the **`hour`**, **`minute`** or **`second`**.

Method **`SetTime`** (lines 48–54) now uses properties **`Hour`**, **`Minute`** and **`Second`** to ensure that instance variables **`hour`**, **`minute`** and **`second`** have valid values. After we define a property, we can use it in the same way that we use a variable. We assign values to properties using the **`=`** (assignment) operator. When this assignment occurs, the code in the **`set`** accessor for that property executes. The reserved word **`value`** represents the argument to the **`set`** accessor. Similarly, methods **`ToUniversalString`** (102–106) and **`ToStandardString`** (109–114) now use properties **`Hour`**, **`Minute`** and **`Second`** to obtain the values of instance variables **`hour`**, **`minute`** and **`second`**. Referencing the property executes the **`get`** accessor for that property.

When we use **`set`** and **`get`** accessor methods throughout the constructors and other methods of class **`Time3`**, we minimize the changes that we must make to the class definition in the event that we alter the data representation from **`hour`**, **`minute`** and **`second`** to another representation (such as total elapsed seconds in the day). When such changes are made, we must provide only new **`set`** and **`get`** accessor bodies. Using this technique also enables programmers to change the implementation of a class without affecting the clients of that class (as long as all the **`public`** methods of the class still are called in the same way).

Software Engineering Observation 8.15

*Accessing **`private`** data through **`set`** and **`get`** accessors not only protects the instance variables from receiving invalid values, but also hides the internal representation of the instance variables from that class's clients. Thus, if representation of the data changes (typically, to reduce the amount of required storage or to improve performance), only the method implementations need to change—the client implementations need not change, as long as the interface provided by the methods is preserved.*

Class **`TimeTest3`** (Fig. 8.7) defines an application with a GUI for manipulating an object of class **`Time3`**. [*Note*: We do not show Visual Studio's *Windows Form Designer* generated code. Instead, line 45 provides a comment to indicate where the generated code appears in the source code file. You can view this code on the CD that accompanies this book.]

```
1  // Fig. 8.7: TimeTest3.cs
2  // Demonstrating Time3 properties Hour, Minute and Second.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
```

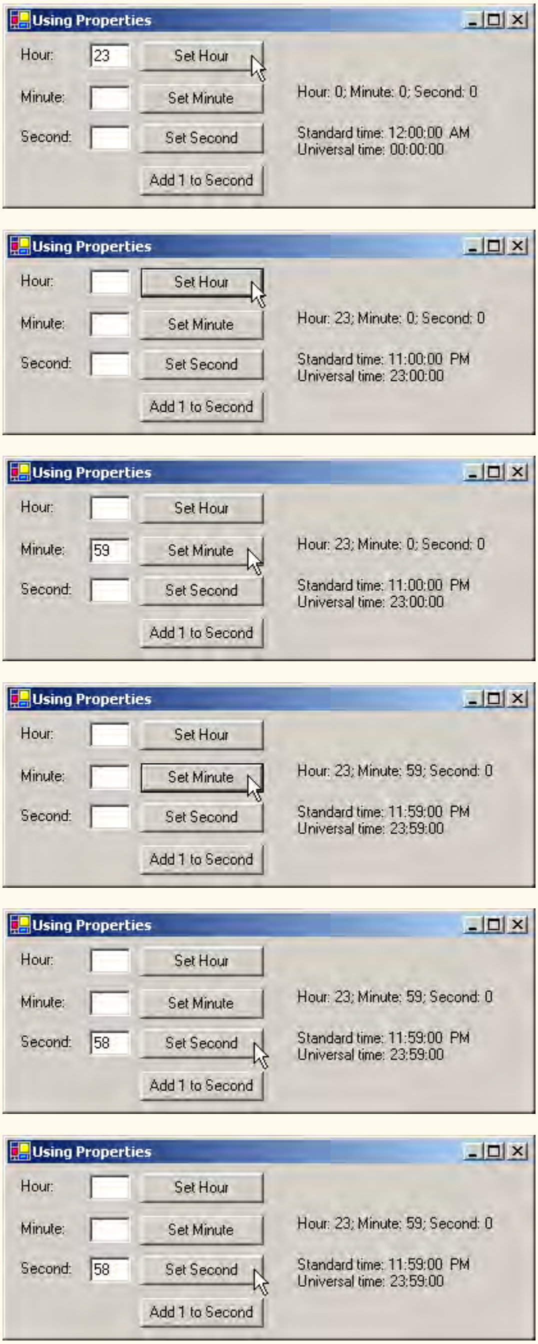
Fig. 8.7 Properties demonstration for class **`Time3`**. (Part 1 of 5.)

```
using System.Windows.Forms;
using System.Data;

// TimeTest3 class definition
public class TimeTest3 : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label hourLabel;
   private System.Windows.Forms.TextBox hourTextBox;
   private System.Windows.Forms.Button hourButton;

   private System.Windows.Forms.Label minuteLabel;
   private System.Windows.Forms.TextBox minuteTextBox;
   private System.Windows.Forms.Button minuteButton;

   private System.Windows.Forms.Label secondLabel;
   private System.Windows.Forms.TextBox secondTextBox;
   private System.Windows.Forms.Button secondButton;

   private System.Windows.Forms.Button addButton;

   private System.Windows.Forms.Label displayLabel1;
   private System.Windows.Forms.Label displayLabel2;

   // required designer variable
   private System.ComponentModel.Container components = null;

   private Time3 time;

   public TimeTest3()
   {
      // Required for Windows Form Designer support
      InitializeComponent();

      time = new Time3();
      UpdateDisplay();
   }

   // Visual Studio .NET generated code

   // main entry point for application
   [STAThread]
   static void Main()
   {
      Application.Run( new TimeTest3() );
   }

   // update display labels
   public void UpdateDisplay()
   {
      displayLabel1.Text = "Hour: " + time.Hour +
         "; Minute: " + time.Minute +
         "; Second: " + time.Second;
```

Fig. 8.7 Properties demonstration for class `Time3`. (Part 2 of 5.)

```
         displayLabel2.Text = "Standard time: " +
            time.ToStandardString() + "\nUniversal time: " +
            time.ToUniversalString();
      }

      // set Hour property when hourButton pressed
      private void hourButton_Click(
         object sender, System.EventArgs e )
      {
         time.Hour = Int32.Parse( hourTextBox.Text );
         hourTextBox.Text = "";
         UpdateDisplay();
      }

      // set Minute property when minuteButton pressed
      private void minuteButton_Click(
         object sender, System.EventArgs e )
      {
         time.Minute = Int32.Parse( minuteTextBox.Text );
         minuteTextBox.Text = "";
         UpdateDisplay();
      }

      // set Second property when secondButton pressed
      private void secondButton_Click(
         object sender, System.EventArgs e )
      {
         time.Second = Int32.Parse( secondTextBox.Text );
         secondTextBox.Text = "";
         UpdateDisplay();
      }

      // add one to Second when addButton pressed
      private void addButton_Click(
         object sender, System.EventArgs e )
      {
         time.Second = ( time.Second + 1 ) % 60;

         if ( time.Second == 0 )
         {
            time.Minute = ( time.Minute + 1 ) % 60;

            if ( time.Minute == 0 )
               time.Hour = ( time.Hour + 1 ) % 24;
         }

         UpdateDisplay();

      } // end method addButton_Click

   } // end class TimeTest3
```

Fig. 8.7 Properties demonstration for class **`Time3`**. (Part 3 of 5.)


Using Properties
Hour:
Set Hour
Minute:
Set Minute
Second:
Set Second
Add 1 to Second

Hour: 23

Hour: 0; Minute: 0; Second: 0

Standard time: 12:00:00 AM
Universal time: 00:00:00

Hour: 23; Minute: 0; Second: 0

Standard time: 11:00:00 PM
Universal time: 23:00:00

Minute: 59

Hour: 23; Minute: 0; Second: 0

Standard time: 11:00:00 PM
Universal time: 23:00:00

Hour: 23; Minute: 59; Second: 0

Standard time: 11:59:00 PM
Universal time: 23:59:00

Second: 58

Hour: 23; Minute: 59; Second: 0

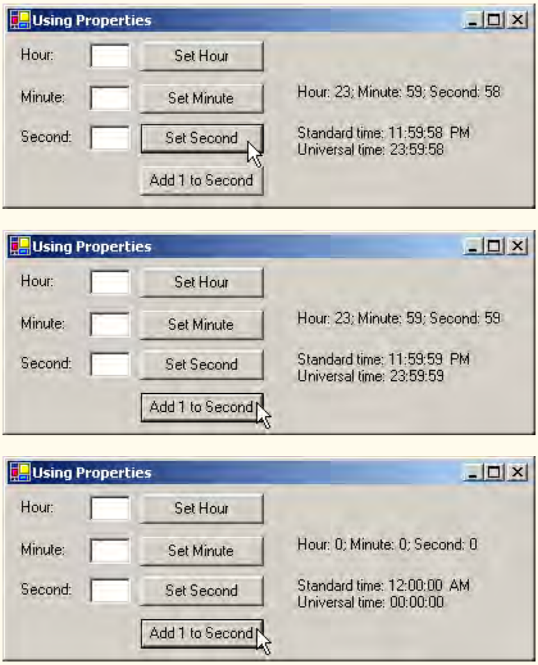
Standard time: 11:59:00 PM
Universal time: 23:59:00

Second: 58

Hour: 23; Minute: 59; Second: 0

Standard time: 11:59:00 PM
Universal time: 23:59:00

Fig. 8.7 Properties demonstration for class **`Time3`**. (Part 4 of 5.)



```
Hour: 23; Minute: 59; Second: 58

Standard time: 11:59:58 PM
Universal time: 23:59:58
```

```
Hour: 23; Minute: 59; Second: 59

Standard time: 11:59:59 PM
Universal time: 23:59:59
```

```
Hour: 0; Minute: 0; Second: 0

Standard time: 12:00:00 AM
Universal time: 00:00:00
```

Fig. 8.7 Properties demonstration for class `Time3`. (Part 5 of 5.)

Line 34 declares **`Time3`** reference **`time`**. Line 41 in the constructor creates an object of class **`Time3`** and assigns it to **`time`**. The GUI contains three text fields in which the user can input values for the **`Time3`** object's **`hour`**, **`minute`** and **`second`** variables, respectively. Next to each text field is a button the user can click to set the value of a particular **`Time3`** property. Lines 66–90 declare three event-handling methods for the buttons' **`Click`** events. Each event handler alters the values a **`Time3`** property (**`Hour`**, **`Minute`** or **`Second`**). The GUI also contains a button that enables the user to increment the **`second`** value by **`1`**. Using the **`Time3`** object's properties, method **`addButton_Click`** (lines 93–108) determines and sets the new time. For example, **`23:59:59`** becomes **`00:00:00`** when the user presses the button. Each modification of the time results in a call to **`UpdateDisplay`**, which uses the **`Time3`** properties to display the **`hour`**, **`minute`** and **`second`** values, and also displays the universal- and standard-time representations.

Properties are not limited to accessing **`private`** data—properties also can be used to calculate values associated with an object. One example of this would be a **`student`** object with a property representing the student's GPA (called **`GPA`**). Programmers can either provide code that calculates the student's GPA in the **`get`** accessor for this property, or they can simply return a **`private`** variable containing the GPA, called **`gpa`**. (The value in this variable will need to be calculated in some other way, such as using a **`CalculateGPA`** method.) The programmer can use either technique, but we recommend using a property that calculates the GPA. Remember that client code should not be required to tell the **`student`** object when to calculate the GPA. The client code simply should use the GPA property. The client should not be aware of the underlying implementation.

8.8 Composition: Objects References as Instance Variables of Other Classes

In many situations, referencing existing objects is more convenient than rewriting the objects' code for new classes in new projects. Suppose we were to implement an **`AlarmClock`** object that needs to know when to sound its alarm. Referencing an existing **`Time`** object (like those from earlier examples in this chapter) is easier than writing a new **`Time`** object. The use of references to objects of preexisting classes as members of new objects is called *composition* (or *aggregation*).

Software Engineering Observation 8.16

One form of software reuse is composition, in which a class has as members references to objects of other classes.

The application of Fig. 8.8, Fig. 8.9 and Fig. 8.10 demonstrates composition. The program contains three classes. Class **`Date`** (Fig. 8.8) encapsulates information relating to a specific date. Class **`Employee`** (Fig. 8.9) encapsulates the name of the employee and two **`Date`** objects representing the **`Employee`**'s birthday and hire date. Class **`CompositionTest`** (Fig. 8.10) creates an object of class **`Employee`** to demonstrate composition.

Class **`Date`** declares **`int`** instance variables **`month`**, **`day`** and **`year`** (lines 9–11). Lines 16–32 define the constructor, which receives values for **`month`**, **`day`** and **`year`** as arguments and assigns these values to the instance variables after ensuring that the values are in a consistent state. Note that lines 25–26 print an error message if the constructor receives an invalid month value. Ordinarily, rather than printing error messages, a constructor would "throw an exception." We discuss exceptions in Chapter 11, Exception Handling. Method **`ToDateString`** (lines 58–61) returns the string representation of a **`Date`**.

```
 1  // Fig. 8.8: Date.cs
 2  // Date class definition encapsulates month, day and year.
 3
 4  using System;
 5
 6  // Date class definition
 7  public class Date
 8  {
 9     private int month;  // 1-12
10     private int day;    // 1-31 based on month
11     private int year;   // any year
12
13     // constructor confirms proper value for month;
14     // call method CheckDay to confirm proper
15     // value for day
16     public Date( int theMonth, int theDay, int theYear )
17     {
```

Fig. 8.8 **`Date`** class encapsulates day, month and year information. (Part 1 of 2.)

```
         // validate month
         if ( theMonth > 0 && theMonth <= 12 )
            month = theMonth;

         else
         {
            month = 1;
            Console.WriteLine(
               "Month {0} invalid. Set to month 1.", theMonth );
         }

         year = theYear;                   // could validate year
         day = CheckDay( theDay );         // validate day

      } // end Date constructor

      // utility method confirms proper day value
      // based on month and year
      private int CheckDay( int testDay )
      {
         int[] daysPerMonth =
            { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

         // check if day in range for month
         if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
            return testDay;

         // check for leap year
         if ( month == 2 && testDay == 29 &&
              ( year % 400 == 0 ||
                ( year % 4 == 0 && year % 100 != 0 ) ) )
            return testDay;

         Console.WriteLine(
            "Day {0} invalid. Set to day 1.", testDay );

         return 1;  // leave object in consistent state
      }

      // return date string as month/day/year
      public string ToDateString()
      {
         return month + "/" + day + "/" + year;
      }

   }  // end class Date
```

Fig. 8.8 **`Date`** class encapsulates day, month and year information. (Part 2 of 2.)

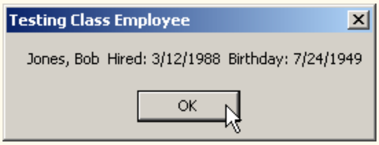
Class **`Employee`** (Fig. 8.9) encapsulates information relating to an employee's birthday and hire date (lines 10–13) using instance variables **`firstName`**, **`lastName`**, **`birthDate`** and **`hireDate`**. Members' **`birthDate`** and **`hireDate`** are references to **`Date`** objects, each of which contains instance variables **`month`**, **`day`** and **`year`**. In this example, class **`Employee`** is *composed of* two references of type **`string`** and two references of class **`Date`**. The **`Employee`** constructor (lines 16–27) takes eight arguments (**`first`**, **`last`**, **`birthMonth`**, **`birthDay`**, **`birthYear`**, **`hireMonth`**, **`hireDay`** and **`hireYear`**). Line 24 passes arguments **`birthMonth`**, **`birthDay`** and **`birthYear`** to the **`Date`** constructor to create the **`birthDate`** object. Similarly, line 25 passes arguments **`hireMonth`**, **`hireDay`** and **`hireYear`** to the **`Date`** constructor to create the **`hireDate`** object. Method **`ToEmployeeString`** (lines 30–35) returns a **`string`** containing the name of the **`Employee`** and the string representations of the **`Employee`**'s **`birthDate`** and **`hireDate`**.

Class **`CompositionTest`** (Fig. 8.10) runs the application with method **`Main`**. Lines 13–14 instantiate an **`Employee`** object and lines 16–17 display the string representation of the **`Employee`** to the user.

```
 1  // Fig. 8.9: Employee.cs
 2  // Employee class definition encapsulates employee's first name,
 3  // last name, birth date and hire date.
 4
 5  using System;
 6
 7  // Employee class definition
 8  public class Employee
 9  {
10     private string firstName;
11     private string lastName;
12     private Date birthDate;  // reference to a Date object
13     private Date hireDate;   // reference to a Date object
14
15     // constructor initializes name, birth date and hire date
16     public Employee( string first, string last,
17        int birthMonth, int birthDay, int birthYear,
18        int hireMonth, int hireDay, int hireYear )
19     {
20        firstName = first;
21        lastName = last;
22
23        // create new Date objects
24        birthDate = new Date( birthMonth, birthDay, birthYear );
25        hireDate = new Date( hireMonth, hireDay, hireYear );
26
27     } // end Employee constructor
28
29     // convert Employee to String format
30     public string ToEmployeeString()
31     {
32        return lastName + ", " + firstName +
33           "  Hired: " + hireDate.ToDateString() +
34           "  Birthday: " + birthDate.ToDateString();
35     }
36
37  }  // end class Employee
```

Fig. 8.9 **`Employee`** class encapsulates employee name, birthday and hire date.

```
// Fig. 8.10: CompositionTest.cs
// Demonstrate an object with member object reference.

using System;
using System.Windows.Forms;

// Composition class definition
class CompositionTest
{
   // main entry point for application
   static void Main( string[] args )
   {
      Employee e =
         new Employee( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );

      MessageBox.Show( e.ToEmployeeString(),
         "Testing Class Employee" );

   } // end method Main

} // end class CompositionTest
```



Jones, Bob Hired: 3/12/1988 Birthday: 7/24/1949

Fig. 8.10 Composition demonstration.

8.9 Using the `this` Reference

Every object can access a reference to itself, called the ***this** reference*. The **`this`** reference can refer implicitly to the instance variables, properties and methods of an object. Keyword **`this`** is commonly used within methods, where **`this`** is a reference to the object on which the method is performing operations. In the Windows application of Fig. 6.4, there are several uses of **`this`** in method **`InitializeComponent`**. The application uses the **`this`** keyword to reference the form that is being initialized. Every form has an **`InitializeComponent`** methods, so the **`this`** reference provides us with an easy way to access the information in the current object. Additional examples of **`this`** appear in Chapters 12 and 13.

We now demonstrate implicit and explicit use of the **`this`** reference to display the **`private`** data of a **`Time4`** object. Class **`Time4`** (Fig. 8.11) defines three **`private`** instance variables—**`hour`**, **`minute`** and **`second`** (lines 9–11). The constructor (lines 14–19) receives three **`int`** arguments to initialize a **`Time4`** object. Note that, for this example, we have made the parameter names for the constructor (line 14) identical to the instance variable names for the class (lines 9–11). We did this to illustrate explicit use of the **`this`** reference. If a method contains a local variable with the same name as an instance variable of that class, that method will refer to the local variable, rather than to the instance variable (i.e., the local variable hides the instance variable in that method's scope). However, the method can use the **`this`** reference to refer to the hidden instance variables explicitly (lines 16–18).

```
 1  // Fig. 8.11: Time4.cs
 2  // Class Time4 provides overloaded constructors.
 3
 4  using System;
 5
 6  // Time4 class definition
 7  public class Time4
 8  {
 9     private int hour;   // 0-23
10     private int minute; // 0-59
11     private int second; // 0-59
12
13     // constructor
14     public Time4( int hour, int minute, int second )
15     {
16        this.hour = hour;
17        this.minute = minute;
18        this.second = second;
19     }
20
21     // create string using this and implicit references
22     public string BuildString()
23     {
24        return "this.ToStandardString(): " +
25           this.ToStandardString() +
26           "\nToStandardString(): " + ToStandardString();
27     }
28
29     // convert time to standard-time (12-hour) format string
30     public string ToStandardString()
31     {
32        return String.Format( "{0}:{1:D2}:{2:D2} {3}",
33           ( ( this.hour == 12 || this.hour == 0 ) ? 12 :
34           this.hour % 12 ), this.minute, this.second,
35           ( this.hour < 12 ? "AM" : "PM" ) );
36     }
37
38  } // end class Time4
```

Fig. 8.11 **this** reference used implicitly and explicitly to enable an object to manipulate its own data and invoke its own methods. (Part 1 of 2)

Method **BuildString** (lines 22–27) returns a **string** created by a statement that uses the **this** reference explicitly and implicitly. Line 25 uses the **this** reference explicitly to call method **ToStandardString**, whereas line 26 uses the **this** reference implicitly to call the same method. Note that both lines perform the same task. Therefore, programmers usually do not use the **this** reference explicitly to reference methods within the current object.

Common Programming Error 8.4

For a method in which a parameter (or local variable) has the same name as an instance variable, use reference **this** *if you wish to access the instance variable; otherwise, the method parameter (or local variable) will be referenced.*

Testing and Debugging Tip 8.2

Avoid method-parameter names (or local variable names) that conflict with instance variable names to prevent subtle, hard-to-trace bugs.

Good Programming Practice 8.6

The explicit use of the **`this`** *reference can increase program clarity in some contexts where* **`this`** *is optional.*

Class **`ThisTest`** (Fig. 8.12) runs the application that demonstrates explicit use of the **`this`** reference. Line 13 instantiates an instance of class **`Time4`**. Lines 15–16 invoke method **`BuildString`** of the **`Time4`** object, then display the results to the user in a **`MessageBox`**.

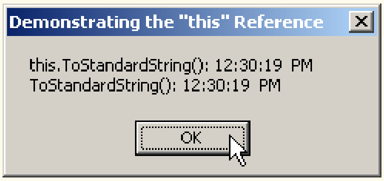
The problem of parameters (or local variables) hiding instance variables can be solved by using properties. If we have a property **`Hour`** that accesses the **`hour`** instance variable, then we would not need to use **`this.hour`** to distinguish between a parameter (or local variable) **`hour`** and the instance variable **`hour`**—we would simply assign **`hour`** to **`Hour`**.

8.10 Garbage Collection

In previous examples, we have seen how a constructor initializes data in an object of a class after the object is created. Operator **`new`** allocates memory for the object, then calls that object's constructor. The constructor might acquire other system resources, such as network connections and databases or files. Objects must have a disciplined way to return memory and release resources when the program no longer uses those objects. Failure to release such resources causes *resource leaks*—potentially exhausting the pool of available resources that programs might need to continue executing.

```
 1  // Fig. 8.12: ThisTest.cs
 2  // Using the this reference.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // ThisTest class definition
 8  class ThisTest
 9  {
10     // main entry point for application
11     static void Main( string[] args )
12     {
13        Time4 time = new Time4( 12, 30, 19 );
14
15        MessageBox.Show( time.BuildString(),
16           "Demonstrating the \"this\" Reference" );
17     }
18  }
```

Fig. 8.12 **`this`** reference demonstration. (Part 1 of 2.)



```
this.ToStandardString(): 12:30:19 PM
ToStandardString(): 12:30:19 PM
```

Fig. 8.12 `this` reference demonstration. (Part 2 of 2.)

Unlike C and C++, in which programmers must manage memory explicitly, C# performs memory management internally. The .NET Framework performs *garbage collection* of memory to return to the system memory that is no longer needed. When the *garbage collector* executes, it locates objects for which the application has no references. Such objects can be collected at that time or during a subsequent execution of the garbage collector. Therefore, the *memory leaks* that are common in such languages as C and C++, where memory is not reclaimed automatically, are rare in C#.

Allocation and deallocation of other resources, such as network connections, database connections and files, must be handled explicitly by the programmer. One technique employed to handle these resources (in conjunction with the garbage collector) is to define a *destructor* (sometimes known as a *finalizer*) that returns resources to the system. The garbage collector calls an object's destructor to perform *termination housekeeping* on that object just before the garbage collector reclaims the object's memory (called *finalization*).

Each class can contain only one destructor. The name of a destructor is formed by preceding the class name with a `~` character. For example, the destructor for class `Time` would be `~Time()`. Destructors do not receive arguments, so destructors cannot be overloaded. When the garbage collector is removing an object from memory, the garbage collector first invokes that object's destructor to clean up resources used by the class. However, we cannot determine exactly when the destructor is called, because we cannot determine exactly when garbage collection occurs. At program termination, any objects that have not been not garbage collected previously will receive destructor calls.

8.11 `static` Class Members

Each object of a class has its own copy of all the instance variables of the class. However, in certain cases, all class objects should share only one copy of a particular variable. Such variables are called ***`static`** variables*. A program contains only one copy of each of a class's `static` variables in memory, no matter how many objects of the class have been instantiated. A `static` variable represents *class-wide information*—all class objects share the same `static` data item.

The declaration of a `static` member begins with the keyword `static`. A `static` variable can be initialized in its declaration by following the variable name with an `=` and an initial value. In cases where a `static` variable requires more complex initialization, programmers can define a ***`static`** constructor* to initialize only the `static` members. Such constructors are optional and must be declared with the `static` keyword, followed by the name of the class. `static` constructors are called before any `static` members are used and before any class objects are instantiated.

We now consider a video-game example to justify the need for `static` class-wide data. Suppose that we have a video game involving `Martian`s and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that there are at least four other `Martian`s present. If there are fewer than five `Martian`s present, each `Martian` becomes cowardly. For this reason, each `Martian` must know the `martianCount`. We could endow class `Martian` with `martianCount` as instance data. However, if we were to do this, then every `Martian` would have a separate copy of the instance data, and, every time we create a `Martian`, we would have to update the instance variable `martianCount` in every `Martian`. The redundant copies waste space, and updating those copies is time-consuming. Instead, we declare `martianCount` to be `static` so that `martianCount` is class-wide data. Each `Martian` can see the `martianCount` as if it were instance data of that `Martian`, but C# maintains only one copy of the `static` variable `martianCount` to save space. This technique also saves time; because there is only one copy, we do not have to increment separate copies of `martianCount` for each `Martian` object.

Performance Tip 8.2

When a single copy of the data will suffice, use `static` *variables to save storage.*

Although `static` variables might seem like *global variables* (variables that can be referenced anywhere in a program) in other programming languages, `static` variables need not be globally accessible. `static` variables have class scope.

The `public static` data members of a class can be accessed through the class name using the dot operator (e.g., `Math.PI`). The `private static` members can be accessed only through methods or properties of the class. `static` members are available as soon as the class is loaded into memory at execution time and they exist for the duration of program execution, even when no objects of that class exist. To enable a program to access a `private static` member when no objects of the class exist, the class must provide a `public static` method or property.

A `static` method cannot access instance (non-`static`) members. Unlike instance methods, a `static` method has no `this` reference, because `static` variables and `static` methods exist independently of any class objects, even when there are no objects of that class.

Common Programming Error 8.5

Using the `this` *reference in a* `static` *method or* `static` *property is a compilation error.*

Common Programming Error 8.6

A call to an instance method or an attempt to access an instance variable from a `static` *method is a compilation error.*

Class `Employee` (Fig. 8.13) demonstrates a `public static` property that enables a program to obtain the value of a `private static` variable. The `static` variable `count` (line 11) is not initialized explicitly, so it receives the value zero by default. Class variable `count` maintains a count of the number of objects of class `Employee` that have been instantiated, including those objects that have already been marked for garbage collection, but have not yet been reclaimed by the garbage collector.

```
// Fig. 8.13: Employee.cs
// Employee class contains static data and a static method.

using System;

// Employee class definition
public class Employee
{
   private string firstName;
   private string lastName;
   private static int count;  // Employee objects in memory

   // constructor increments static Employee count
   public Employee( string fName, string lName )
   {
      firstName = fName;
      lastName = lName;

      ++count;

      Console.WriteLine( "Employee object constructor: " +
       firstName + " " + lastName + "; count = " + Count );
   }

   // destructor decrements static Employee count
   ~Employee()
   {
      --count;

      Console.WriteLine( "Employee object destructor: " +
         firstName + " " + lastName + "; count = " + Count );
   }

   // FirstName property
   public string FirstName
   {
      get
      {
         return firstName;
      }
   }

   // LastName property
   public string LastName
   {
      get
      {
         return lastName;
      }
   }

```

Fig. 8.13 `static` members are accessible to all objects of a class. (Part 1 of 2.)

```
52     // static Count property
53     public static int Count
54     {
55        get
56        {
57           return count;
58        }
59     }
60
61  } // end class Employee
```

Fig. 8.13 `static` members are accessible to all objects of a class. (Part 2 of 2.)

When objects of class `Employee` exist, `static` member `count` can be used in any method of an `Employee` object—in this example, the constructor (lines 14–23) increments `count`, and the destructor (lines 26–32) decrements `count`. If no objects of class `Employee` exist, the value of member `count` can be obtained through `static` property `Count` (lines 53–59); this also works when there are `Employee` objects in memory.

Class `StaticTest` (Fig. 8.14) runs the application that demonstrates the `static` members of class `Employee` (Fig. 8.13). Lines 12–13 use the `static` property `Count` of class `Employee` to obtain the current `count` value before the program creates `Employee` objects. Notice that the syntax used to access a `static` member is:

ClassName`.`*StaticMember*

On line 13, *ClassName* is `Employee` and *StaticMember* is `Count`. Recall that we used this syntax in prior examples to call the `static` methods of class `Math` (e.g., `Math.Pow`, `Math.Abs`, etc.) and other methods, such as `Int32.Parse` and `MessageBox.Show`.

Next, lines 16–17 instantiate two `Employee` objects and assign them to references `employee1` and `employee2`. Each call to the `Employee` constructor increments the `count` value by one. Lines 19–26 display the value of `Count` as well as the names of the two employees. Lines 30–31 set references `employee1` and `employee2` to `null`, so they no longer refer to the `Employee` objects. Because these were the only references in the program to the `Employee` objects, those objects can now be garbage collected.

```
 1  // Fig. 8.14: StaticTest.cs
 2  // Demonstrating static class members.
 3
 4  using System;
 5
 6  // StaticTest class definition
 7  class StaticTest
 8  {
 9     // main entry point for application
10     static void Main( string[] args )
11     {
12        Console.WriteLine( "Employees before instantiation: " +
13           Employee.Count + "\n" );
14
```

Fig. 8.14 `static` member demonstration. (Part 1 of 2.)

```
15        // create two Employees
16        Employee employee1 = new Employee( "Susan", "Baker" );
17        Employee employee2 = new Employee( "Bob", "Jones" );
18
19        Console.WriteLine( "\nEmployees after instantiation: " +
20           "Employee.Count = " + Employee.Count + "\n" );
21
22        // display the Employees
23        Console.WriteLine( "Employee 1: " +
24           employee1.FirstName + " " + employee1.LastName +
25           "\nEmployee 2: " + employee2.FirstName +
26           " " + employee2.LastName + "\n" );
27
28        // remove references to objects to indicate that
29        // objects can be garbage collected
30        employee1 = null;
31        employee2 = null;
32
33        // force garbage collection
34        System.GC.Collect();
35
36        // wait until collection completes
37        System.GC.WaitForPendingFinalizers();
38
39        Console.WriteLine(
40           "\nEmployees after garbage collection: " +
41           Employee.Count );
42     }
43  }
```

```
Employees before instantiation: 0

Employee object constructor: Susan Baker; count = 1
Employee object constructor: Bob Jones; count = 2

Employees after instantiation: Employee.Count = 2

Employee 1: Susan Baker
Employee 2: Bob Jones

Employee object destructor: Bob Jones; count = 1
Employee object destructor: Susan Baker; count = 0

Employees after garbage collection: 0
```

Fig. 8.14 `static` member demonstration. (Part 2 of 2.)

The garbage collector is not invoked directly by the program. Either the garbage collector reclaims the memory for objects when the runtime determines garbage collection is appropriate, or the operating system recovers the memory when the program terminates. However, it is possible to request that the garbage collector attempt to collect available objects. Line 34 uses **`public static`** method ***`Collect`*** from class ***`GC`*** (namespace

`System`) to make this request. The garbage collector is not guaranteed to collect all objects that are currently available for collection. If the garbage collector decides to collect objects, the garbage collector first invokes the destructor of each object. It is important to understand that the garbage collector executes as an independent entity called a *thread*. (Threads are discussed in Chapter 14, Multithreading.) It is possible for multiple threads to execute in parallel on a multiprocessor system or to share a processor on a single-processor system. Thus, a program could run in parallel with garbage collection. For this reason, we call `static` method *`WaitForPendingFinalizers`* of class `GC` (line 37), which forces the program to wait until the garbage collector invokes the destructors for all objects that are ready for collection and reclaims those objects. When the program reaches lines 41, we are assured that both destructor calls completed and that the value of `count` has been decremented accordingly.

In this example, the output shows that the destructor was called for each `Employee`, which decrements the `count` value by two (once per `Employee` being collected). Lines 39–41 use property `Count` to obtain the value of `count` after invoking the garbage collector. If the objects had not been collected, the `count` would be greater than zero.

Toward the end of the output, notice that the `Employee` object for `Bob Jones` was finalized before the `Employee` object for `Susan Baker`. However, the output of this program on your system could differ. The garbage collector is not guaranteed to collect objects in a specific order.

8.12 `const` and `readonly` Members

C# allows programmers to create *constants* whose values cannot change during program execution.

Testing and Debugging Tip 8.3

If a variable should never change, make it a constant. This helps eliminate errors that might occur if the value of the variable were to change.

To create a constant data member of a class, declare that member using either the *`const`* or *`readonly`* keyword. Data members declared as `const` implicitly are `static` and must be initialized in their declaration. Data members declared as `readonly` can be initialized in their declaration or in their class's constructor. Neither `const` nor `readonly` values can be modified once they are initialized, except that `readonly` variables can be assigned values in several constructors (only one of which will be called when an object is initialized).

Common Programming Error 8.7

Declaring a class data member as `const` but failing to initialize it in that class's declaration is a syntax error.

Common Programming Error 8.8

Assigning a value to a `const` variable after that variable is initialized is a compilation error.

Common Programming Error 8.9

The declaration of a `const` member as `static` is a syntax error, because a `const` member implicitly is `static`.

Common Programming Error 8.10

Declaring a class data member as **`readonly`** *and attempting to use it before it is initialized is a logic error.*

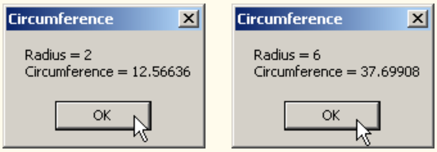
Members that are declared as **`const`** must be assigned values at compile time. Therefore, **`const`** members can be initialized only with other constant values, such as integers, string literals, characters and other **`const`** members. Constant members with values that cannot be determined at compile time must be declared with keyword **`readonly`**. We mentioned previously that a **`readonly`** member can be assigned a value only once, either when it is declared or within the constructor of the class. When initializing a **`static readonly`** member in a constructor, a **`static`** constructor must be used.

Figure 8.15 demonstrates constants. The program consists of two classes—class **`Constants`** (lines 8–22) defines two constants, and class **`UsingConstAndReadonly`** (lines 25–43) demonstrates the constants in class **`Constants`**.

```
 1  // Fig. 8.15: UsingConstAndReadOnly.cs
 2  // Demonstrating constant values with const and readonly.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // Constants class definition
 8  public class Constants
 9  {
10     // create constant PI
11     public const double PI = 3.14159;
12
13     // radius is a constant
14     // that is uninitialized
15     public readonly int radius;
16
17     public Constants( int radiusValue )
18     {
19        radius = radiusValue;
20     }
21
22  } // end class Constants
23
24  // UsingConstAndReadOnly class definition
25  public class UsingConstAndReadonly
26  {
27     // method Main creates Constants
28     // object and displays its values
29     static void Main( string[] args )
30     {
31        Random random = new Random();
32
33        Constants constantValues =
34           new Constants( random.Next( 1, 20 ) );
35
```

Fig. 8.15 **`const`** and **`readonly`** class member demonstration. (Part 1 of 2.)

```
36         MessageBox.Show( "Radius = " + constantValues.radius +
37            "\nCircumference = " +
38            2 * Constants.PI * constantValues.radius,
39            "Circumference" );
40
41      } // end method Main
42
43   } // end class UsingConstAndReadOnly
```



Radius = 2
Circumference = 12.56636

Radius = 6
Circumference = 37.69908

Fig. 8.15 `const` and `readonly` class member demonstration. (Part 2 of 2.)

Line 11 in class `Constants` creates constant `PI` using keyword `const` and initializes `PI` with the `double` value `3.14159`—an approximation of π that the program uses to calculate the circumferences of circles. Note that we could have used the predefined constant `PI` of class `Math` (`Math.PI`) as the value, but we wanted to demonstrate how to define a `const` variable explicitly. The compiler must be able to determine a `const` variable's value at compile time; otherwise, a compilation error will occur. For example, if line 11 initialized `PI` with the expression:

```
Double.Parse( "3.14159" )
```

the compiler would generate an error. Although the expression uses `string` literal `"3.14159"` (a constant value) as an argument, the compiler cannot evaluate the method call `Double.Parse` at compile time.

Variables declared `readonly` can be initialized at execution time. Line 15 declares `readonly` variable `radius`, but does not initialize it. The `Constants` constructor (lines 17–20) receives an `int` value and assigns it to `radius` when the program creates a `Constants` object. Note that radius also can be initialized with a more complex expression, such as a method call that returns an `int`.

Class `UsingConstAndReadonly` (lines 25–43) uses the `const` and `readonly` variables of class `Constants`. Lines 33–34 use a `Random` object to generate a random `int` between `1` and `20` that corresponds to a circle's `radius`, then pass that value to the `Constants` constructor to initialize the `readonly` variable `radius`. Lines 36–39 output the radius and circumference of a circle in a `MessageBox`. Line 36 uses `Constants`'s reference `contantValues` to access `readonly` variable `radius`. Line 38 computes the circle's circumference using `const` variable `Constants.PI` and `readonly` variable `radius`. Note that we use `static` syntax to access `const` variable `PI`, because `const` variables implicitly are `static`.

8.13 Indexers

Sometimes a class encapsulates data that a program can manipulate as a list of elements. Such a class can define special properties called *indexers* that allow array-style indexed access to lists of elements. With "conventional" C# arrays, the subscript number must be an integer value. A benefit of indexers is that the programmer can define both integer subscripts and non-integer subscripts. For example, a programmer could allow client code to manipulate data using **`string`**s as subscripts that represent the data items' names or descriptions. When manipulating "conventional" C# array elements, the array subscript operator always returns the same data type—i.e., the type of the array. Indexers are more flexible—they can return any data type, even one that is different from the type of the data in the list of elements.

Although an indexer's subscript operator is used like an array-subscript operator, indexers are defined as properties in a class. Unlike normal properties, for which the programmer can choose an appropriate property name, indexers must be defined with keyword **`this`**. Indexers have the general form:

```
accessModifier returnType this[ IndexType1 name1, IndexType2 name2, … ]
{
   get
   {
      // use name1, name2, ... here to get data
   }

   set
   {
      // use name1, name2, ... here to set data
   }
}
```

The *IndexType* parameters specified in the brackets (**`[]`**) are accessible to the **`get`** and **`set`** accessors. These accessors define how to use the index (or indices) to select or modify the appropriate data member. As with properties, **`get`** must **`return`** a value of type *returnType* and **`set`** can use the **`value`** keyword to reference the value that should be assigned to the data member.

Common Programming Error 8.11

*Declaring indexers as **`static`** is a syntax error.*

The program of Fig. 8.16 contains two classes—class **`Box`** (lines 14–74) represents a box with a length, a width and a height, and class **`IndexerTest`** (lines 77–177) demonstrates class **`Box`**'s indexers.

```
1  // Fig. 8.16: IndexerTest.cs
2  // Indexers provide access to an object's members via a
3  // subscript operator.
4
5  using System;
6  using System.Drawing;
7  using System.Collections;
8  using System.ComponentModel;
9  using System.Windows.Forms;
10 using System.Data;
11
```

Fig. 8.16 Indexers provide subscripted access to an object's members. (Part 1 of 6.)

```
// Box class definition represents a box with length,
// width and height dimensions
public class Box
{
   private string[] names = { "length", "width", "height" };
   private double[] dimensions = new double[ 3 ];

   // constructor
   public Box( double length, double width, double height )
   {
      dimensions[ 0 ] = length;
      dimensions[ 1 ] = width;
      dimensions[ 2 ] = height;
   }

   // access dimensions by integer index number
   public double this[ int index ]
   {
      get
      {
         return ( index < 0 || index >= dimensions.Length ) ?
            -1 : dimensions[ index ];
      }

      set
      {
         if ( index >= 0 && index < dimensions.Length )
            dimensions[ index ] = value;
      }

   } // end numeric indexer

   // access dimensions by their string names
   public double this[ string name ]
   {
      get
      {
         // locate element to get
         int i = 0;

         while ( i < names.Length &&
            name.ToLower() != names[ i ] )
            i++;

         return ( i == names.Length ) ? -1 : dimensions[ i ];
      }

      set
      {
         // locate element to set
         int i = 0;

```

Fig. 8.16 Indexers provide subscripted access to an object's members. (Part 2 of 6.)

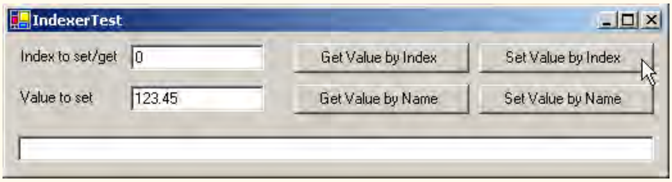
```
            while ( i < names.Length &&
               name.ToLower() != names[ i ] )
               i++;

            if ( i != names.Length )
               dimensions[ i ] = value;
         }

      } // end indexer

   } // end class Box

   // Class IndexerTest
   public class IndexerTest : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Label indexLabel;
      private System.Windows.Forms.Label nameLabel;

      private System.Windows.Forms.TextBox indexTextBox;
      private System.Windows.Forms.TextBox valueTextBox;

      private System.Windows.Forms.Button nameSetButton;
      private System.Windows.Forms.Button nameGetButton;

      private System.Windows.Forms.Button intSetButton;
      private System.Windows.Forms.Button intGetButton;

      private System.Windows.Forms.TextBox resultTextBox;

      // required designer variable
      private System.ComponentModel.Container components = null;

      private Box box;

      // constructor
      public IndexerTest()
      {
         // required for Windows Form Designer support
         InitializeComponent();

         // create block
         box = new Box( 0.0, 0.0, 0.0 );
      }

      // Visual Studio .NET generated code

      // main entry point for application
      [STAThread]
      static void Main()
      {
         Application.Run( new IndexerTest() );
      }

```

Fig. 8.16 Indexers provide subscripted access to an object's members. (Part 3 of 6.)

```
      // display value at specified index number
      private void ShowValueAtIndex( string prefix, int index )
      {
         resultTextBox.Text =
            prefix + "box[ " + index + " ] = " + box[ index ];
      }

      // display value with specified name
      private void ShowValueAtIndex( string prefix, string name )
      {
         resultTextBox.Text =
            prefix + "box[ " + name + " ] = " + box[ name ];
      }

      // clear indexTextBox and valueTextBox
      private void ClearTextBoxes()
      {
         indexTextBox.Text = "";
         valueTextBox.Text = "";
      }

      // get value at specified index
      private void intGetButton_Click(
         object sender, System.EventArgs e )
      {
         ShowValueAtIndex(
            "get: ", Int32.Parse( indexTextBox.Text ) );
         ClearTextBoxes();
      }

      // set value at specified index
      private void intSetButton_Click(
         object sender, System.EventArgs e )
      {
         int index = Int32.Parse( indexTextBox.Text );
         box[ index ] = Double.Parse( valueTextBox.Text );

         ShowValueAtIndex( "set: ", index );
         ClearTextBoxes();
      }

      // get value with specified name
      private void nameGetButton_Click(
         object sender, System.EventArgs e )
      {
         ShowValueAtIndex( "get: ", indexTextBox.Text );
         ClearTextBoxes();
      }

      // set value with specified name
      private void nameSetButton_Click(
         object sender, System.EventArgs e )
      {
```

Fig. 8.16 Indexers provide subscripted access to an object's members. (Part 4 of 6.)

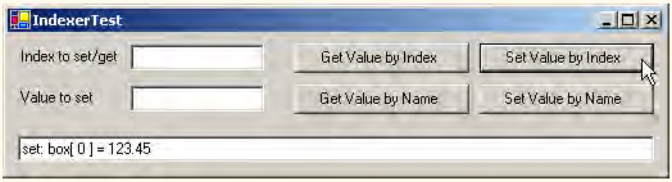
```
      box[ indexTextBox.Text ] =
         Double.Parse( valueTextBox.Text );

      ShowValueAtIndex( "set: ", indexTextBox.Text );
      ClearTextBoxes();
   }

} // end class IndexerTest
```


IndexerTest
Get Value by Index
Set Value by Index
Get Value by Name
Set Value by Name

Index to set/get: 0

Value to set: 123.45

Before setting value by index number

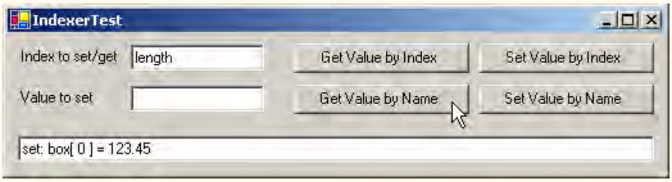

IndexerTest
Get Value by Index
Set Value by Index
Get Value by Name
Set Value by Name

Index to set/get:

Value to set:

set: box[0] = 123.45

After setting value by index number

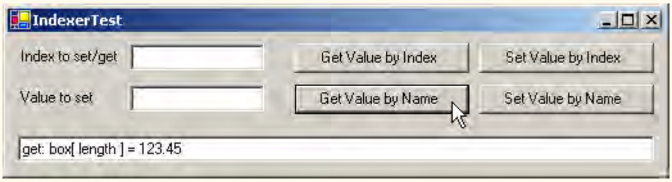

IndexerTest
Get Value by Index
Set Value by Index
Get Value by Name
Set Value by Name

Index to set/get: length

Value to set:

set: box[0] = 123.45

Before getting value by dimension name

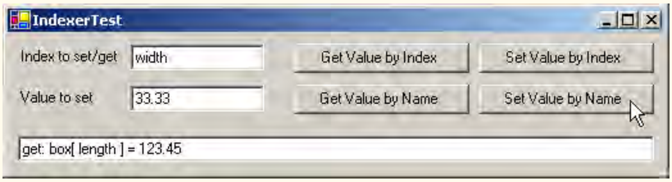

IndexerTest
Get Value by Index
Set Value by Index
Get Value by Name
Set Value by Name

Index to set/get:

Value to set:

get: box[length] = 123.45

After getting value by dimension name


IndexerTest
Get Value by Index
Set Value by Index
Get Value by Name
Set Value by Name

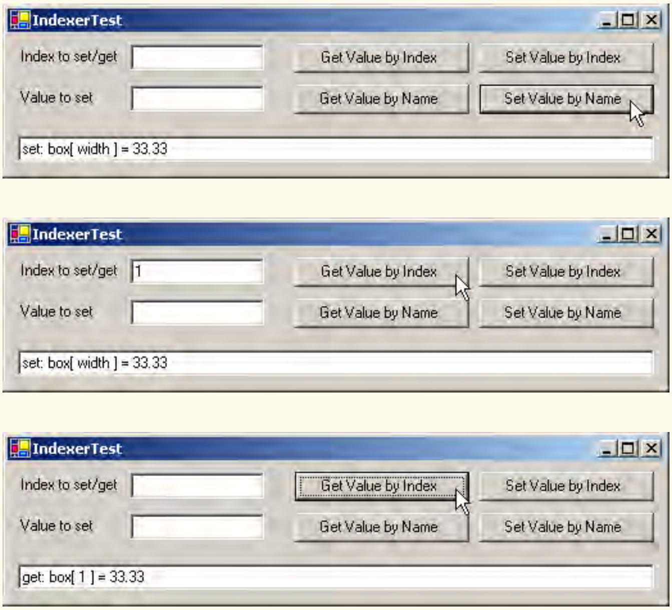
Index to set/get: width

Value to set: 33.33

get: box[length] = 123.45

Before setting value by dimension name

Fig. 8.16 Indexers provide subscripted access to an object's members. (Part 5 of 6.)



set: box[width] = 33.33

After setting value by dimension name

Index to set/get: 1

set: box[width] = 33.33

Before getting value by index number

get: box[1] = 33.33

After getting value by index number

Fig. 8.16 Indexers provide subscripted access to an object's members. (Part 6 of 6.)

The `private` data members of class `Box` are `string` array `names` (line 16), which contains the names (i.e., `"length"`, `"width"` and `"height"`) for the dimensions of a `Box`, and `double` array `dimensions` (line 17), which contains the size of each dimension. Each element in array `names` corresponds to an element in array `dimensions` (e.g., `dimensions[ 2 ]` contains the height of the `Box`).

`Box` defines two indexers (lines 28–42 and lines 45–72) that each `return` a `double` value representing the size of the dimension specified by the indexer's parameter. Indexers can be overloaded like methods. The first indexer uses an `int` subscript to manipulate an element in the `dimensions` array. The second indexer uses a `string` subscript representing the name of the dimension to manipulate an element in the `dimensions` array. Each indexer returns `-1` if its `get` accessor encounters an invalid subscript. Each indexer's `set` accessor assigns `value` to the appropriate element of `dimensions` only if the index is valid. Normally, the programmer would have an indexer throw an exception if an indexer received an invalid index. We discuss how to throw exceptions in Chapter 11, Exception Handling.

Notice that the `string` indexer uses a `while` structure to search for a matching `string` in the `names` array (lines 64–66). If a match is found, the indexer manipulates the corresponding element in array `dimensions` (line 69).

Class `IndexerTest` is a `System.Windows.Forms.Form` that manipulates the `private` data members of class `Box` through `Box`'s indexers. Instance variable `box` is declared at line 96 and initialized in the constructor at line 105 with dimensions of `0.0`. The event handler for button **Get Value by Index** (lines 139–145) invokes method `ShowValueAtIndex` (lines 118–122) to retrieve the value at the index number specified in `indexTextBox`. The event handler for button **Set Value by Index** (lines 148–156) assigns the value in `valueTextBox` to the location specified in `indexTextBox`. The event handler for button **Get Value by Name** (159–164) invokes the overloaded method `ShowValueAtIndex` (lines 125–129) to retrieve the value with the name specified in `valueTextBox`. The event handler for button **Set Value by Name** (lines 167–175) assigns the value in `valueTextBox` to the location with the name specified in `indexTextBox`.

8.14 Data Abstraction and Information Hiding

As we pointed out at the beginning of this chapter, classes normally hide the details of their implementation from their clients. This is called *information hiding*. As an example of information hiding, let us consider a data structure called a *stack*.

Students can think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it is always placed at the top (referred to as *pushing* the dish onto the stack). Similarly, when a dish is removed from the pile, it is always removed from the top (referred to as *popping* the dish off the stack). Stacks are known as *last-in, first-out (LIFO) data structures*—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

Stacks can be implemented with arrays and with other data structures, such as linked lists. (We discuss stacks and linked lists in Chapter 23, Data Structures.) A client of a stack class need not be concerned with the stack's implementation. The client knows only that when data items are placed in the stack, these items will be recalled in last-in, first-out order. The client cares about *what* functionality a stack offers, but not about *how* that functionality is implemented. This concept is referred to as *data abstraction*. Although programmers might know the details of a class's implementation, they should not write code that depends on these details. This enables a particular class (such as one that implements a stack and its operations, *push* and *pop*) to be replaced with another version without affecting the rest of the system. As long as the `public` services of the class do not change (i.e., every method still has the same name, return type and parameter list in the new class definition), the rest of the system is not affected.

Most programming languages emphasize actions. In these languages, data exist to support the actions that programs must take. Data are "less interesting" than actions. Data are "crude." Only a few built-in data types exist, and it is difficult for programmers to create their own data types. C# and the object-oriented style of programming elevate the importance of data. The primary activities of object-oriented programming in C# is the creation of data types (i.e., classes) and the expression of the interactions among objects of those data types. To create languages that emphasize data, the programming-languages community needed to formalize some notions about data. The formalization we consider here is the notion of *abstract data types (ADTs)*. ADTs receive as much attention today as structured programming did decades earlier. ADTs, however, do not replace structured programming. Rather, they provide an additional formalization to improve the program-development process.

Consider built-in type `int`, which most people would associate with an integer in mathematics. Rather, an `int` is an abstract representation of an integer. Unlike mathematical integers, computer `int`s are fixed in size. For example, type `int` in .NET is limited approximately to the range –2 billion to +2 billion. If the result of a calculation falls outside this range, an error occurs, and the computer responds in some machine-dependent manner.

It might, for example, “quietly” produce an incorrect result. Mathematical integers do not have this problem. Therefore, the notion of a computer **`int`** is only an approximation of the notion of a real-world integer. The same is true of **`float`** and other built-in types.

We have taken the notion of **`int`** for granted until this point, but we now consider it from a new perspective. Types like **`int`**, **`float`**, **`char`** and others are all examples of abstract data types. These types are representations of real-world notions to some satisfactory level of precision within a computer system.

An ADT actually captures two notions: A *data representation* and the *operations* that can be performed on that data. For example, in C#, an **`int`** contains an integer value (data) and provides addition, subtraction, multiplication, division and modulus operations; however, division by zero is undefined. C# programmers use classes to implement abstract data types.

Software Engineering Observation 8.17

Programmers can create types through the use of the class mechanism. These new types can be designed so that they are as convenient to use as the built-in types. This marks C# as an extensible language. *Although the language is easy to extend via new types, the programmer cannot alter the base language itself.*

Another abstract data type we discuss is a *queue*, which is similar to a “waiting line.” Computer systems use many queues internally. A queue offers well-understood behavior to its clients: Clients place items in a queue one at a time via an *enqueue* operation, then get those items back one at a time via a *dequeue* operation. A queue returns items in *first-in, first-out (FIFO)* order, which means that the first item inserted in a queue is the first item removed. Conceptually, a queue can become infinitely long, but real queues are finite.

The queue hides an internal data representation that keeps track of the items currently waiting in line, and it offers a set of operations to its clients (*enqueue* and *dequeue*). The clients are not concerned about the implementation of the queue—clients simply depend upon the queue to operate “as advertised.” When a client enqueues an item, the queue should accept that item and place it in some kind of internal FIFO data structure. Similarly, when the client wants the next item from the front of the queue, the queue should remove the item from its internal representation and deliver the item in FIFO order (i.e., the item that has been in the queue the longest should be the next one returned by the next dequeue operation).

The queue ADT guarantees the integrity of its internal data structure. Clients cannot manipulate this data structure directly—only the queue ADT has access to its internal data. Clients are able to perform only allowable operations on the data representation; the ADT rejects operations that its public interface does not provide.

8.15 Software Reusability

C# programmers concentrate both on crafting new classes and on reusing classes from the Framework Class Library (FCL), which contains thousands of predefined classes. Developers construct software by combining programmer-defined classes with well-defined, carefully tested, well-documented, portable and widely available FCL classes. This kind of software reusability speeds the development of powerful, high-quality software. *Rapid applications development (RAD)* is of great interest today.

The FCL allows C# programmers to achieve software reusability across platforms that support .NET and rapid applications development. C# programmers focus on the high-level programming issues and leave the low-level implementation details to classes in the FCL. For example, a C# programmer who writes a graphics program does not need to know the details of every .NET-platform graphics capability. Instead, C# programmers concentrate on learning and using the FCL's graphics classes.

The FCL enables C# developers to build applications faster by reusing preexisting, extensively tested classes. In addition to reducing development time, FCL classes also improve programmers' abilities to debug and maintain applications, because proven software compenents are being used. For programmers to take advantage of the FCL's classes, they must familiarize themselves with the FCL's rich set of capabilities.

Software reuse is not limited to Windows-application development. The FCL also includes classes for creating *Web services*, which are applications packaged as services that clients can access via the Internet. Any C# application is a potential Web service, so C# programmers can reuse existing applications as building blocks to form larger more sophisticated Web-enabled applications.

Many people believe that Web services represent the next phase in the evolution of software development, in which the Web provides a library of functionality from which developers can build applications in a platform-independent manner. As Microsoft's premier .NET language, C# provides all the features necessary for creating scalable, robust Web services. We formally introduce Web Services in Chapter 21, ASP .NET and Web Services.

8.16 Namespaces and Assemblies

As we have seen in almost every example in the text, classes from preexisting libraries, such as the .NET Framework, must be imported into a C# program by adding a reference to the appropriate libraries (a process we demonstrated in Section 3.2). Remember that each class in the Framework Class Library belongs to a specific namespace. The preexisting code in the FCL facilitates software reuse.

Programmers should concentrate on making the software components they create reusable. However, doing so often results in *naming collisions*. For example, two classes defined by different programmers can have the same name. If a program needs both of those classes, the program must have a way to distinguish between the two classes in the code.

Common Programming Error 8.12

Attempting to compile code that contains naming collisions will generate compilation errors.

Namespaces help minimize this problem by providing a convention for *unique class names*. No two classes in a given namespace can have the same name, but different namespaces can contain classes of the same name. With hundreds of thousands of people writing C# programs, there is a good chance the names that one programmer chooses to describe classes will conflict with the names that other programmers choose for their classes.

We begin our discussion of reusing existing class definitions in Fig. 8.17, which provides the code for class **`Time3`** (originally defined in Fig. 8.6). When reusing class definitions between programs, programmers create class libraries that can be imported for use in a program via a **`using`** statement. Only **`public`** classes can be reused from class libraries. Non-**`public`** classes can be used only by other classes in the same assembly.

The only difference between class **`Time3`** in this example and the version in Fig. 8.6 is that we show the **`namespace`**, i.e., **`TimeLibrary`**, in which **`Time3`** is defined. Each class library is defined in a **`namespace`** that contains all the classes in the library. We will demonstrate momentarily how to package class **`Time3`** into **`TimeLibrary.dll`**—the *dynamic link library* that we create for reuse in other programs. Programs can load dynamic link libraries at execution time to access common functionality that can be shared among many programs. A dynamic link library represents an assembly. When a project uses a class library, the project must contain a reference to the assembly that defines the class library.

```
// Fig. 8.17: TimeLibrary.cs
// Placing class Time3 in an assembly for reuse.

using System;

namespace TimeLibrary   // specifies namespace for class Time3
{
   // Time3 class definition
   public class Time3
   {
      private int hour;    // 0-23
      private int minute;  // 0-59
      private int second;  // 0-59

      // Time3 constructor initializes instance variables to
      // zero to set default time to midnight
      public Time3()
      {
         SetTime( 0, 0, 0 );
      }

      // Time3 constructor: hour supplied, minute and second
      // defaulted to 0
      public Time3( int hour )
      {
         SetTime( hour, 0, 0 );
      }

      // Time3 constructor: hour and minute supplied, second
      // defaulted to 0
      public Time3( int hour, int minute )
      {
         SetTime( hour, minute, 0 );
      }

      // Time3 constructor: hour, minute and second supplied
      public Time3( int hour, int minute, int second )
      {
         SetTime( hour, minute, second );
      }

```

Fig. 8.17 Assembly **`TimeLibrary`** contains class **`Time3`**. (Part 1 of 3.)

```
      // Time3 constructor: initialize using another Time3 object
      public Time3( Time3 time )
      {
         SetTime( time.Hour, time.Minute, time.Second );
      }

      // Set new time value in 24-hour format. Perform validity
      // checks on the data. Set invalid values to zero.
      public void SetTime(
         int hourValue, int minuteValue, int secondValue )
      {
         Hour = hourValue;
         Minute = minuteValue;
         Second = secondValue;
      }

      // property Hour
      public int Hour
      {
         get
         {
            return hour;
         }

         set
         {
            hour = ( ( value >= 0 && value < 24 ) ? value : 0 );
         }

      } // end property Hour

      // property Minute
      public int Minute
      {
         get
         {
            return minute;
         }

         set
         {
            minute = ( ( value >= 0 && value < 60 ) ? value : 0 );
         }

      } // end property Minute

      // property Second
      public int Second
      {
         get
         {
            return second;
         }
```

Fig. 8.17 Assembly **`TimeLibrary`** contains class **`Time3`**. (Part 2 of 3.)

```
95
96           set
97           {
98              second = ( ( value >= 0 && value < 60 ) ? value : 0 );
99           }
100
101       } // end property Second
102
103       // convert time to universal-time (24 hour) format string
104       public string ToUniversalString()
105       {
106          return String.Format(
107             "{0:D2}:{1:D2}:{2:D2}", Hour, Minute, Second );
108       }
109
110       // convert time to standard-time (12 hour) format string
111       public string ToStandardString()
112       {
113          return String.Format( "{0}:{1:D2}:{2:D2} {3}",
114             ( ( Hour == 12 || Hour == 0 ) ? 12 : Hour % 12 ),
115             Minute, Second, ( Hour < 12 ? "AM" : "PM" ) );
116       }
117
118    } // end class Time3
119 }
```

Fig. 8.17 Assembly **`TimeLibrary`** contains class **`Time3`**. (Part 3 of 3.)

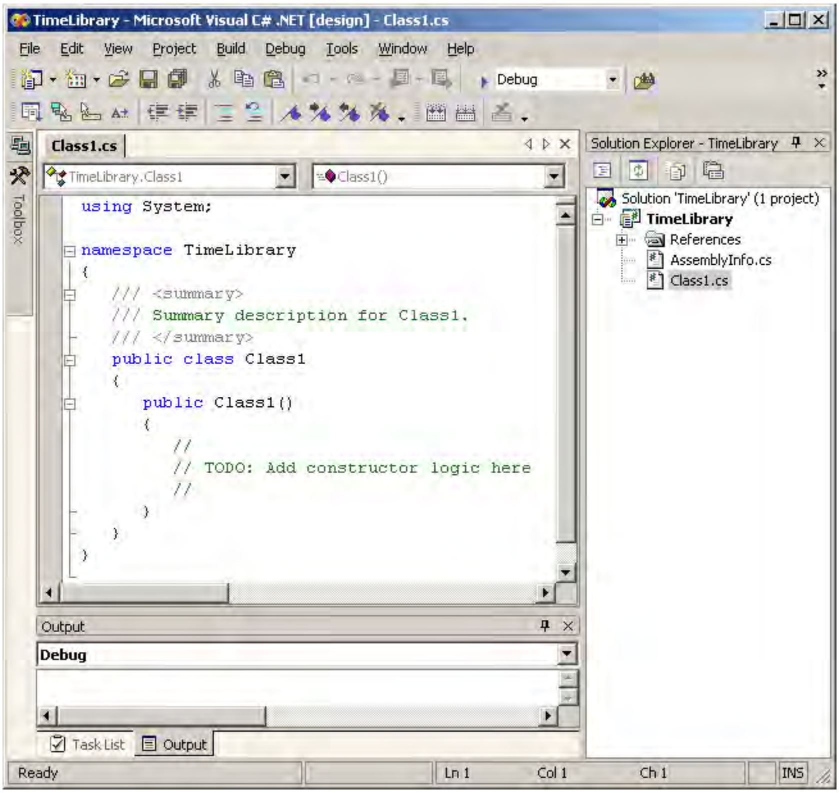
We now describe, step-by-step, how to create the class library **`TimeLibrary`** containing class **`Time3`**:

1. *Create a class library project.* From the **File** menu, choose option **New**, followed by **Project…**. In the **New Project** dialog, ensure that **C# Projects** is selected in the **Project Types** section and click **Class Library**. Name the project **TimeLibrary** and choose the directory in which you would like to store the project. A simple class library will be created, as shown in Fig. 8.18. There are two important points to note about the generated code. The first is that the class does not contain a **`Main`** method. This indicates that the class in the class library cannot be used to begin the execution of an application. This class is designed to be used by other programs. Also notice that **`Class1`** is created as a **`public`** class. If another project uses this library, only the library's **`public`** classes are accessible. We created class **`Time3`** as **`public`** for this purpose (line 9 of Fig. 8.17) by renaming the class **`Class1`** (created by Visual Studio as part of the project) to **`Time3`**. In the **Solution Explorer**, we also renamed the **`Class1.cs`** file as **`Time3.cs`**.
2. *Add the code for class **`Time3`**.* Delete the code for the **`Class1`** constructor. Then, copy the remainder of the **`Time3`** code (lines 11–116) from Fig. 8.17 (you can find this file in the examples on the CD that accompanies this book) and paste the code in the body of the class definition shown in Fig. 8.18.
3. *Compile the code.* From the **Build** menu, choose option **Build Solution**. The code should compile successfully. Remember that this code cannot be executed—

there is no entry point into the program. In fact, if you try running the program by selecting the **Debug** menu and choosing **Start**, Visual Studio .NET displays an error message.

Compiling the project creates an assembly (a dynamic link library) that represents the new class library. This assembly can be found in the `bin\Debug` directory of the project. By default, the assembly name will include the namespace name. (In this case, the name will be `TimeLibrary.dll`.) The assembly file contains class `Time3`, which other projects can use. Assembly files, which have file extensions `.dll` and `.exe`, are integral to C#. The Windows operating system uses executable files (`.exe`) to run applications, whereas it uses library files (`.dll`, or *dynamic link library*) to represent code libraries that can be loaded dynamically by many applications and shared among those applications.

Next, we define a console application project containing class `AssemblyTest` (Fig. 8.19), which uses class `Time3` in assembly `TimeLibrary.dll` to create a `Time3` object and display its standard and universal string formats.


TimeLibrary - Microsoft Visual C# .NET [design] - Class1.cs
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug
Class1.cs
TimeLibrary.Class1
Class1()
Toolbox
Solution Explorer - TimeLibrary
Solution 'TimeLibrary' (1 project)
TimeLibrary
References
AssemblyInfo.cs
Class1.cs
Output
Debug
Task List
Output
Ready
Ln 1
Col 1
Ch 1
INS

```
using System;

namespace TimeLibrary
{
   /// <summary>
   /// Summary description for Class1.
   /// </summary>
   public class Class1
   {
      public Class1()
      {
         //
         // TODO: Add constructor logic here
         //
      }
   }
}
```

Fig. 8.18 Simple Class Library.

```
 1  // Fig. 8.19: AssemblyTest.cs
 2  // Using class Time3 from assembly TimeLibrary.
 3
 4  using System;
 5  using TimeLibrary;
 6
 7  // AssemblyTest class definition
 8  class AssemblyTest
 9  {
10     // main entry point for application
11     static void Main( string[] args )
12     {
13        Time3 time = new Time3( 13, 27, 6 );
14
15        Console.WriteLine(
16           "Standard time: {0}\nUniversal time: {1}\n",
17           time.ToStandardString(), time.ToUniversalString() );
18     }
19  }
```

```
Standard time: 1:27:06 PM
Universal time: 13:27:06
```

Fig. 8.19 Assembly **`TimeLibrary`** used from class **`AssemblyTest`**.

Before class **`AssemblyTest`** can use class **`Time3`**, the project containing class **`AssemblyTest`** must have a reference to the **`TimeLibrary`** assembly. To add the reference, select **Add Reference** from the **Project** menu. Using the **Browse** button, select **`TimeLibrary.dll`** (located in the **`bin\Debug`** directory of the **`TimeLibrary`** project), then click **OK** to add the resource to the project. After adding the reference, use keyword **`using`** to inform the compiler that we will use classes from namespace **`TimeLibrary`** (line 5 in Fig. 8.19).

8.17 Class View and Object Browser

Now that we have introduced key concepts of object-based programming, we present two features that Visual Studio provides to facilitate the design of object-oriented applications—***Class View*** and ***Object Browser***.

The **Class View** displays the variables and methods for all classes in a project. To access this feature, select **Class View** from the **View** menu. Figure 8.20 shows the **Class View** for the **`TimeTest1`** project of Fig. 8.1 and Fig. 8.2 (class **`Time1`** and class **`TimeTest1`**). The view follows a hierarchical structure, positioning the project name (**`TimeTest1`**) as the root and including a series of nodes (e.g., classes, variables, methods etc.). If a plus sign (**`+`**) appears to the left of a node, that node can be expanded to show other nodes. By contrast, if a minus sign (**`-`**) appears to the left of a node, that node has been expanded (and can be collapsed). According to the **Class View**, project **`TimeTest`** contains class **`Time1`** and class **`TimeTest1`** as *children*. Class **`Time1`** contains methods **`SetTime`**, **`Time1`**, **`ToStandardString`** and **`ToUniversalString`** (indicated by purple boxes) and instance variables **`hour`**, **`minute`** and **`second`** (indicated by blue

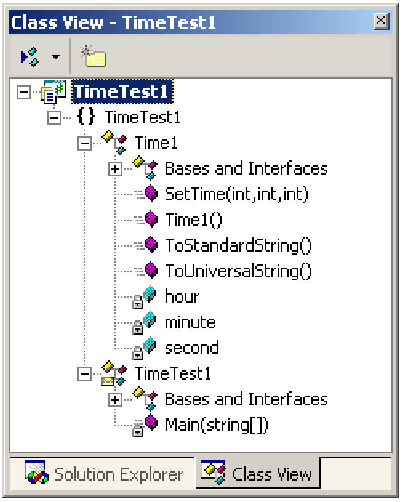
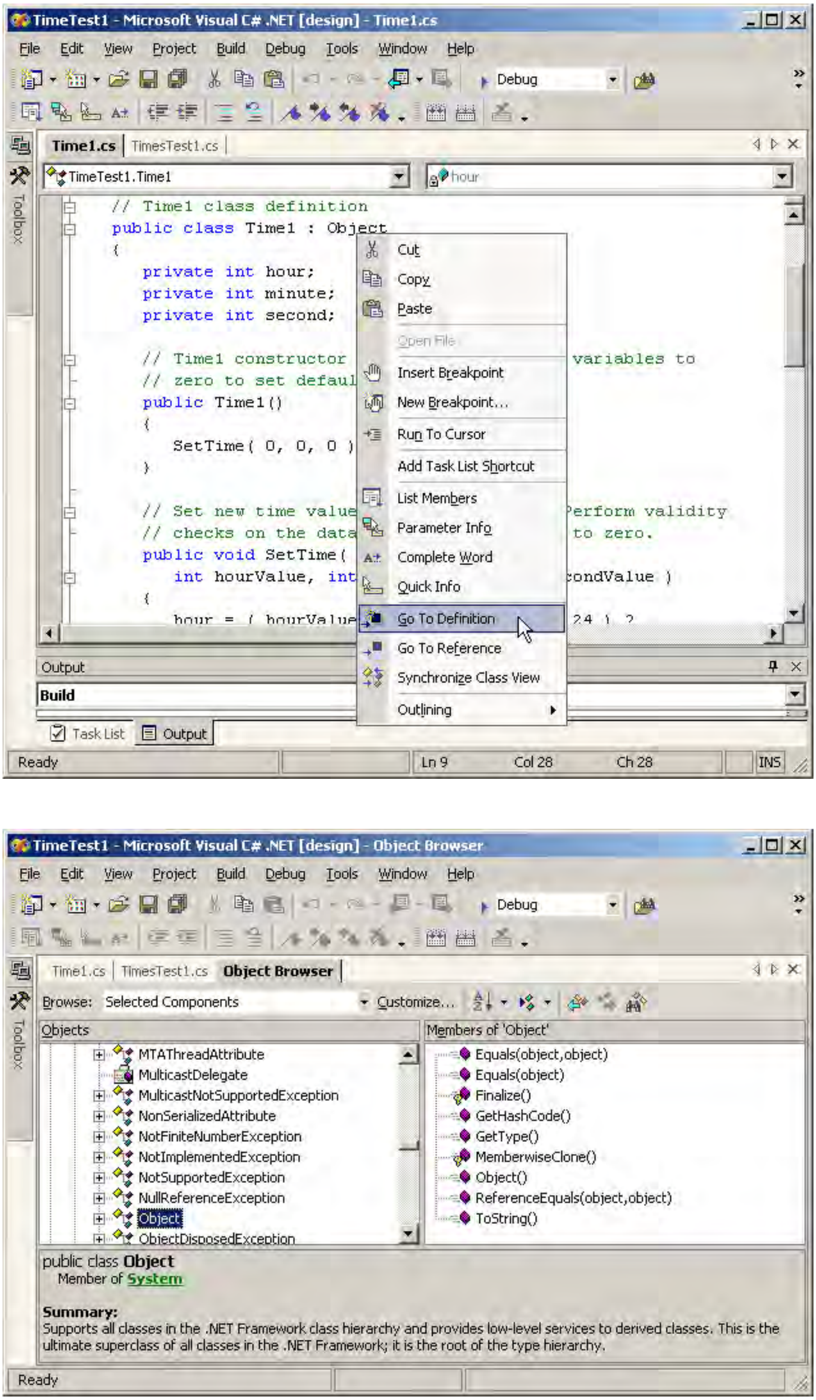


Fig. 8.20 **Class View** of class `Time1` (Fig. 8.1) and class `TimeTest` (Fig. 8.2).

boxes). The lock icons, placed to the left of the blue-box icons for the instance variables, specify that the variables are `private`. Class `TimeTest1` contains method `Main`. Note that both class `Time1` and class `TimeTest1` contain the **Bases and Interfaces** node. If you expand this node, you will see class `Object` in each case, because each class inherits from class `System.Object` (discussed in Chapter 9).

Visual Studio's **Object Browser** lists all classes in the C# library. Developers use the **Object Browser** to learn about the functionality provided by a specific class. To open the **Object Browser**, right click any built-in C# class or method in the code editor and select **Go To Definition**. Figure 8.21 depicts the **Object Browser** when the user right clicks the class name `Object` in the code editor. Note that the **Object Browser** lists all methods provided by class `Object` in the **Members of 'Object'** window—this window offers developers "instant access" to information regarding the functionality of various objects. Note also that the **Object Browser** lists in the **Objects** window all classes in the FCL. The **Object Browser** can be a quick mechanism to learn about a class or method of a class. Remember that you can also view the complete description of a class or method in the online documentation available through the **Help** menu in Visual Studio .NET.

This chapter is the first in a series of three chapters that cover the fundamentals of object-based and object-oriented programming. In this chapter, we discussed how to create proper class definitions, how to control access to class members and several features commonly used to craft valuable classes for reuse by other programmers. Chapter 9, focusses on *inheritance*. In that chapter, you will learn how to build classes that inherit data and functionality from existing class definitions. You also will learn other C# features that are specific to the inheritance relationship between classes. These features serve as the basis for the object-oriented programming concept called *polymorphism* that we present in Chapter 10.


TimeTest1 - Microsoft Visual C# .NET [design] - Time1.cs
File Edit View Project Build Debug Tools Window Help
Debug
Time1.cs
TimesTest1.cs
TimeTest1.Time1
hour
Toolbox
Cut
Copy
Paste
Open File
Insert Breakpoint
New Breakpoint...
Run To Cursor
Add Task List Shortcut
List Members
Parameter Info
Complete Word
Quick Info
Go To Definition
Go To Reference
Synchronize Class View
Outlining
Output
Build
Task List
Output
Ready
Ln 9
Col 28
Ch 28
INS

```
// Time1 class definition
public class Time1 : Object
{
   private int hour;
   private int minute;
   private int second;

   // Time1 constructor                    variables to
   // zero to set defaul
   public Time1()
   {
      SetTime( 0, 0, 0 )
   }

   // Set new time value                  erform validity
   // checks on the data                   to zero.
   public void SetTime(
      int hourValue, int                  condValue )
   {
      hour = ( hourValue                  24 ) ?
```

TimeTest1 - Microsoft Visual C# .NET [design] - Object Browser
File Edit View Project Build Debug Tools Window Help
Debug
Time1.cs
TimesTest1.cs
Object Browser
Toolbox
Browse: Selected Components
Customize...
Objects
MTAThreadAttribute
MulticastDelegate
MulticastNotSupportedException
NonSerializedAttribute
NotFiniteNumberException
NotImplementedException
NotSupportedException
NullReferenceException
Object
ObjectDisposedException
Members of 'Object'
Equals(object,object)
Equals(object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
Object()
ReferenceEquals(object,object)
ToString()
Ready

public class **Object**
Member of System

Summary:
Supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. This is the ultimate superclass of all classes in the .NET Framework; it is the root of the type hierarchy.

Fig. 8.21 **Object Browser** when user selects `Object` from `Time1.cs`.

SUMMARY

- Every class in C# inherits directly or indirectly from class `Object`.
- Keywords `public` and `private` are member access modifiers.
- Instance variables and methods that are declared with member access modifier `public` are accessible wherever the program has a reference to an object of that class.
- Instance variables and methods that are declared with member access modifier `private` are accessible only to non-`static` methods of the class in which the `private` members are defined.
- The `private` methods often are called utility methods, or helper methods, because they can be called only by other methods of that class and are used to support the operation of those methods.
- Access methods can read or display data. Another common use for access methods is to test the truth of conditions—such methods often are called predicate methods.
- A constructor initializes the instance variables of a class object. A class's constructor is called automatically when an object of that class is instantiated.
- It is common to have overloaded constructors for a class. Normally, constructors are `public`.
- Every class in C#, such as the classes from the .NET Framework, belongs to a namespace.
- If the programmer does not specify the namespace for a class, the class is placed in the default namespace, which includes the compiled classes in the current directory.
- Instance variables can be initialized by the class constructor, or they can be assigned values by the `set` accessor of a property.
- Instance variables that are not initialized explicitly by the programmer are initialized by the compiler (primitive numeric variables are set to `0`, `bool` values are set to `false` and references are set to `null`).
- Classes simplify programming, because the client code need only be concerned with the `public` operations encapsulated in an object of the class.
- A class's non-`static` instance variables and methods belong to that class's scope. Within a class's scope, class members are immediately accessible to all of that class's non-`static` methods and can be referenced simply by name. Outside a class's scope, class members cannot be referenced directly by name.
- Variables defined in a method are known only to that method (i.e., they are local to that method). Such variables are said to have block scope.
- If a method defines a variable that has the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in that method.
- To allow clients to manipulate the value of `private` data, the class can provide a property definition, which will enable the user to access this `private` data in a safe way.
- A property definition contains accessor methods that handle the details of modifying and returning data.
- A property definition can contain a `set` accessor, a `get` accessor or both. A `get` accessor enables the client to read the field's value and the `set` accessor enables the client to modify the value.
- When an object is created, its members can be initialized by a constructor of that object's class.
- If no constructors are defined for a class, a default constructor will be provided by the compiler. This constructor contains no code and takes no parameters.
- Methods and constructors of a class can be overloaded. To overload a method of a class, simply provide a separate method definition with the same name for each version of the method. Remember that overloaded methods/constructors must have different parameter lists.

- Although **`set`** and **`get`** accessors can provide access to **`private`** data, the access is restricted by the programmer's implementation of those methods.
- One form of software reuse is composition, in which a class contains as members references to objects of other classes.
- The **`this`** reference is used implicitly and explicitly to refer to both the instance variables and the non-**`static`** methods of an object.
- The .NET Framework performs automatic garbage collection.
- Every class in C# can have a destructor that typically returns resources to the system. The destructor for an object is guaranteed to be called to perform termination housekeeping on the object just before the garbage collector reclaims the memory for the object (called finalization).
- In certain cases, all objects of a class should share only one copy of a particular variable. Programmers use **`static`** variables for this and other reasons.
- A **`static`** variable represents class-wide information—all objects of the class share the same piece of data.
- The declaration of a **`static`** member begins with the keyword **`static`**. Such variables have class scope.
- A class's **`public static`** members can be accessed via the class name and the dot operator (e.g., **`Math.PI`**).
- A class's **`private static`** members can be accessed only through methods or properties of the class.
- A method declared **`static`** cannot access non-**`static`** members.
- C# allows programmers to create constants whose values cannot change during program execution.
- To create a constant member of a C# class, the programmer must declare that member using either the **`const`** or **`readonly`** keyword.
- Members declared **`const`** must be initialized in the declaration; those declared with **`readonly`** can be initialized in the constructor, but must be initialized before they are used.
- Neither **`const`** nor **`readonly`** values can be modified once they are initialized.
- A class can define indexers to provide subscripted access to the data in an object of that class.
- Indexers can be defined to use any data type as the subscript.
- Each indexer can define a **`get`** and **`set`** accessor.
- Classes normally hide their implementation details from the clients of the classes. This is called information hiding.
- C# and the object-oriented style of programming elevate the importance of data. The primary activities of object-oriented programming in C# are the creation of data types (i.e., classes) and the expression of the interactions among objects of those data types.
- C# programmers concentrate on crafting new classes and reusing existing classes.
- Software reusability speeds the development of powerful, high-quality software. Rapid applications development (RAD) is of great interest today.
- Each class and interface in the .NET Framework belongs to a specific namespace (or library) that contains a group of related classes and interfaces. Namespaces provide a mechanism for software reuse.
- There is a good chance that the names you choose for classes will conflict with the names that other programmers choose for their classes. For this reason, namespaces provide a convention for unique class names.

- The Visual Studio .NET **Class View** displays the variables and methods for all classes in a project.
- The Visual Studio .NET **Object Browser** lists all classes in the C# library. Developers use the **Object Browser** to learn about the functionality provided by a specific object.

TERMINOLOGY

abstract data type (ADT)
access method
action
action-oriented
aggregation
assembly
attribute (data)
behavior (method)
block scope
body of a class definition
built-in data types
case sensitivity
class
class definition
class library
class scope
class implements abstract data type
class-scope variable hidden by method-scope variable
Class View
"class-wide" information
client of a class
`Collect` method of `GC`
compile a class
composition
consistent state
constant
constructor
create a code library
create class from existing class definition
create a namespace
create a reusable class
create data types
data abstraction
data in support of actions
data integrity
data member
data representation of an abstract data type
data structure
default constructor
destructor
division by zero is undefined
`.dll`
dot (`.`) operator
dynamic link library
encapsulate
enqueue operation
`.exe`
explicit use of `this` reference
extensible language
finalizer
first-in, first-out (FIFO) data structure
garbage collector
`GC` class
`get` accessor
GUI event handling
helper method
hide an instance variable
hide implementation details
hide internal data representation
implementation
indexer
indexer `get` accessor
indexer `set` accessor
information hiding
inheritance
initial set of classes
initialize to default values
initialize a class object
initialize an instance variable
insert an item into a container object
instance of a built-in type
instance of a user-defined type
instance variable
instantiate (or create) an object
interactions among objects
interface
internal data representation
`IsEmpty`
`IsFull`
last-in, first-out (LIFO) data structure
library
linked list
local variable of a method
member access modifier
memory leak
method overloading
namespace

`new` operator
no-argument constructor
non-`public` method
object (or instance)
Object Browser
`Object` class
object orientation
object passed by reference
“object speak”
“object think”
object-based programming (OBP)
object-oriented programming (OOP)
overloaded constructor
overloaded method
polymorphism
popping off a stack
predicate method
`private` keyword
`private static` member
procedural programming language
program-development process
programmer-defined type
`public` keyword
`public` method
`public` operations encapsulated in an object
`public` service
`public static` member
pushing into a stack
queue
rapid applications development (RAD)
reclaim memory
reference to a new object
resource leak
reusable software component
service of a class
`set` accessor of a property
signature
software reuse
stack
standard-time format
`static` variable
`static` variables have class scope
`static` keyword
structured programming
termination housekeeping
`this` keyword
universal-time format
user-defined type
utility method
validity checking
variable
`WaitForPendingFinalizers` method of class `GC`
waiting line

SELF-REVIEW EXERCISES

8.1 Fill in the blanks in each of the following statements:

a) Client code can access a class’s members via the __________ operator in conjunction with a reference to an object of the class.
b) Members of a class declared __________ are accessible only to methods of the class in which those members are defined.
c) A __________ initializes the instance variables of a class.
d) A property __________ accessor is used to assign values to `private` instance variables of a class.
e) Methods of a class normally are declared __________, and instance variables of a class normally are declared __________.
f) A __________ accessor of a property is used to retrieve values of `private` data of a class.
g) The keyword __________ introduces a class definition.
h) Members of a class declared __________ are accessible anywhere that an object of the class is in scope.
i) The __________ operator allocates memory dynamically for an object of a specified type and returns a __________ to that type.
j) A __________ variable represents class-wide information.
k) The keyword __________ specifies that an object or variable is not modifiable after it is initialized at execution time.
l) A method declared `static` cannot access __________ class members.

8.2 State whether each of the following is *true* of *false*. If *false* explain why.

a) All objects are passed by reference.
b) Constructors can have return values.
c) Properties must define **`get`** and **`set`** accessors.
d) The **`this`** reference of an object is a reference to that object itself.
e) A **`static`** member can be referenced when no object of that type exists.
f) A **`static`** member of a class can be referenced through an instance of the class.
g) Variables declared **`const`** must be initialized either in a declaration or in the class constructor.
h) Different namespaces cannot have classes/methods with the same names.
i) Assembly files are not required to define an entry point (**`Main`** method).
j) Indexers can return any type in C#.

ANSWERS TO SELF-REVIEW EXERCISES

8.1 a) dot (**`.`**). b) **`private`**. c) constructor. d) **`set`**. e) **`public`**, **`private`**. f) **`get`**. g) **`class`**. h) **`public`**. i) **`new`**, reference. j) **`static`**. k) **`readonly`**. l) non-**`static`**.

8.2 a) True. b) False. Constructors are not permitted to return values. c) False. A property definition can specify a **`set`** accessor, a **`get`** accessor or both. d) True. e) True. f) False. A **`static`** member of a class can only be referenced through the class name. g) False. Variables declared **`const`** must be initialized when they are declared. h) False. Different namespaces can have classes/methods with the same names. i) True. j) True.

EXERCISES

8.3 Create a class called **`Complex`** for performing arithmetic with complex numbers. Write a driver program to test your class.

Complex numbers have the form

`realPart + imaginaryPart *` ***i***

where *i* is

$$\sqrt{-1}$$

Use floating-point variables to represent the **`private`** data of the class. Provide a constructor that enables an object of this class to be initialized when it is declared. Provide a no-argument constructor with default values in case no initializers are provided. Provide **`public`** methods for each of the following:

a) *Addition of two* ***`Complex`*** *numbers*. The real parts are added together and the imaginary parts are added together.
b) *Subtraction of two* ***`Complex`*** *numbers*. The real part of the right operand is subtracted from the real part of the left operand and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.
c) *Printing of* ***`Complex`*** *numbers in the form* ***`(a, b)`***, *where* ***`a`*** *is the real part and* ***`b`*** *is the imaginary part.*

8.4 Modify the **`Date`** class of Fig. 8.8 to perform error checking on the initializer values for instance variables **`month`**, **`day`** and **`year`**. Also, provide a method **`NextDay`** to increment the day by one. The **`Date`** object should always remain in a consistent state. Write a program that tests the **`NextDay`** method in a loop that prints the date during each iteration of the loop to illustrate that the **`NextDay`** method works correctly. Be sure to test the following cases:

a) Incrementing into the next month.
b) Incrementing into the next year.

8.5 Create a class **`TicTacToe`** that will enable you to write a complete program to play the game of Tic-Tac-Toe. The class contains as **`private`** data a 3-by-3 double array of characters. The constructor should initialize the empty board to all spaces, **`' '`**. Allow two players. Wherever the first player moves, place an **`'X'`** in the specified square; place an **`'O'`** wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won or if the game is a draw via a **`GameStatus`** method. [*Hint*: use an enumeration constant to return the following statuses: **`WIN`**, **`DRAW`**, **`CONTINUE`**.] Write Windows Application **`TicTacToeTest`** to test your class. If you feel ambitious, modify your program so that the computer makes the moves for one of the players automatically. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional Tic-Tac-Toe on a 4-by-4-by-4 board [Note: This is a challenging project that could take many weeks of effort!]

8.6 Create a **`Date`** class with the following capabilities:

a) Output the date in multiple formats such as

```
MM/DD/YYYY
June 14, 2001
DDD YYYY
```

b) Use overloaded constructors to create **`Date`** objects initialized with dates of the formats in part a).

8.7 Create class **`SavingsAccount`**. Use **`static`** variable **`annualInterestRate`** to store the interest rate for all account holders. Each object of the class contains a **`private`** instance variable **`savingsBalance`** indicating the amount the saver currently has on deposit. Provide method **`CalculateMonthlyInterest`** to calculate the monthly interest by multiplying the **`savingsBalance`** by **`annualInterestRate`** divided by 12; this interest should be added to **`savingsBalance`**. Provide a **`static`** method **`ModifyInterestRate`** that sets the **`annualInterestRate`** to a new value. Write a driver program to test class **`SavingsAccount`**. Instantiate two **`savingsAccount`** objects, **`saver1`** and **`saver2`**, with balances of $2000.00 and $3000.00, respectively. Set **`annualInterestRate`** to 4%, then calculate the monthly interest and print the new balances for each of the savers. Then set the **`annualInterestRate`** to 5% and calculate the next month's interest and print the new balances for each of the savers.

8.8 Write a console application that implements a **`Square`** shape. Class **`Square`** should contain an instance property **`Side`** that has **`get`** and **`set`** accessors for **`private`** data. Provide two constructors: one that takes no arguments and another that takes a **`side`** length as a value. Write an application class that tests class **`Square`**'s functionality.

9

Object-Oriented Programming: Inheritance

Objectives

- To understand inheritance and software reusability.
- To understand the concepts of base classes and derived classes.
- To understand member access modifier **`protected`** and **`internal`**.
- To be able to use the **`base`** reference to access base-class members
- To understand the use of constructors and finalizers in base classes and derived classes.
- To present a case study that demonstrates the mechanics of inheritance.

Say not you know another entirely, till you have divided an inheritance with him.
Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
Bertrand Russell

Good as it is to inherit a library, it is better to collect one.
Augustine Birrell

Outline

9.1 Introduction
9.2 Base Classes and Derived Classes
9.3 `protected` Members
9.4 Relationship between Base Classes and Derived Classes
9.5 Case Study: Three-Level Inheritance Hierarchy
9.6 Constructors and Destructors in Derived Classes
9.7 Software Engineering with Inheritance

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

9.1 Introduction

In this chapter, we begin our discussion of object-oriented programming (OOP) by introducing one of its main features—*inheritance*. Inheritance is a form of software reusability in which classes are created by absorbing an existing class's data and behaviors and embellishing them with new capabilities. Software reusability saves time during program development. It also encourages the reuse of proven and debugged high-quality software, which increases the likelihood that a system will be implemented effectively.

When creating a class, instead of writing completely new instance variables and methods, the programmer can designate that the new class should *inherit* the class variables, properties and methods of another class. The previously defined class is called the *base class*, and the new class is referred to as the *derived class*. (Other programming languages, such as Java, refer to the base class as the *superclass*, and the derived class as the *subclass*.) Once created, each derived class can become the base class for future derived classes. A derived class, to which unique class variables, properties and methods normally are added, is often larger than its base class. Therefore, a derived class is more specific than its base class and represents a more specialized group of objects. Typically, the derived class contains the behaviors of its base class and additional behaviors. The *direct base class* is the base class from which the derived class explicitly inherits. An *indirect base class* is inherited from two or more levels up the *class hierarchy*. In the case of *single inheritance*, a class is derived from one base class. C#, unlike C++, does not support *multiple inheritance* (which occurs when a class is derived from more than one direct base class). (We explain in Chapter 10 how C# can use interfaces to realize many of the benefits of multiple inheritance while avoiding the associated problems.)

Every object of a derived class is also an object of that derived class's base class. However, base-class objects are not objects of their derived classes. For example, all cars are vehicles, but not all vehicles are cars. As we continue our study of object-oriented programming in Chapters 9 and 10, we take advantage of this relationship to perform some interesting manipulations.

Experience in building software systems indicates that significant amounts of code deal with closely related special cases. When programmers are preoccupied with special cases, the details can obscure the "big picture." With object-oriented programming, programmers focus on the commonalities among objects in the system, rather than on the special cases. This process is called *abstraction*.

We distinguish between the *“is-a” relationship* and the *“has-a” relationship*. “Is-a” represents inheritance. In an “is-a” relationship, an object of a derived class also can be treated as an object of its base class. For example, a car *is a* vehicle. By contrast, “has-a” stands for composition (composition is discussed in Chapter 8). In a “has-a” relationship, a class object contains one or more object references as members. For example, a car *has a* steering wheel.

Derived-class methods might require access to their base-class instance variables, properties and methods. A derived class can access the non-**`private`** members of its base class. Base-class members that should not be accessible to properties or methods of a class derived from that base class via inheritance are declared **`private`** in the base class. A derived class can effect state changes in **`private`** base-class members, but only through non-**`private`** methods and properties provided in the base class and inherited into the derived class.

Software Engineering Observation 9.1

Properties and methods of a derived class cannot directly access **`private`** *members of their base class.*

Software Engineering Observation 9.2

Hiding **`private`** *members helps programmers test, debug and correctly modify systems. If a derived class could access its base class’s* **`private`** *members, classes that inherit from that derived class could access that data as well. This would propagate access to what should be* **`private`** *data, and the benefits of information hiding would be lost.*

One problem with inheritance is that a derived class can inherit properties and methods it does not need or should not have. It is the class designer’s responsibility to ensure that the capabilities provided by a class are appropriate for future derived classes. Even when a base-class property or method is appropriate for a derived class, that derived class often requires the property or method to perform its task in a manner specific to the derived class. In such cases, the base-class property or method can be *overridden* (redefined) in the derived class with an appropriate implementation.

New classes can inherit from abundant *class libraries*. Organizations develop their own class libraries and can take advantage of other libraries available worldwide. Someday, the vast majority of new software likely will be constructed from *standardized reusable components*, as most hardware is constructed today. This will facilitate the development of more powerful and abundant software.

9.2 Base Classes and Derived Classes

Often, an object of one class “is an” object of another class, as well. For example, a rectangle *is a* quadrilateral (as are squares, parallelograms and trapezoids). Thus, class **`Rectangle`** can be said to *inherit* from class **`Quadrilateral`**. In this context, class **`Quadrilateral`** is a base class, and class **`Rectangle`** is a derived class. A rectangle *is a* specific type of quadrilateral, but it is incorrect to claim that a quadrilateral *is a* rectangle—the quadrilateral could be a parallelogram or some other type of **`Quadrilateral`**. Figure 9.1 lists several simple examples of base classes and derived classes.

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Fig. 9.1 Inheritance examples.

Every derived-class object "is an" object of its base class, and one base class can have many derived classes; therefore, the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes. For example, the base class **`Vehicle`** represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, derived-class **`Car`** represents only a small subset of all **`Vehicle`**s.

Inheritance relationships form tree-like hierarchical structures. A class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they are employed in inheritance arrangements, they become affiliated with other classes. A class becomes either a base class, supplying data and behaviors to other classes, or a derived class, inheriting its data and behaviors from other classes.

Let us develop a simple inheritance hierarchy. A university community has thousands of members. These members consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teachers. This organizational structure yields the inheritance hierarchy, depicted in Fig. 9.2. Note that the inheritance hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors and seniors. Each arrow in the hierarchy represents an "is-a" relationship. For example, as we follow the arrows in this class hierarchy, we can state, "an **`Employee`** *is a* **`CommunityMember`**" and "a **`Teacher`** *is a* **`Faculty`** member." **`CommunityMember`** is the *direct base class* of **`Employee`**, **`Student`** and **`Alumnus`**. In addition, **`CommunityMember`** is an *indirect base class* of all the other classes in the hierarchy diagram.

Starting from the bottom of the diagram, the reader can follow the arrows and apply the *is-a* relationship to the topmost base class. For example, an **`Administrator`** *is a* **`Faculty`** member, *is an* **`Employee`** and *is a* **`CommunityMember`**. In C#, an **`Administrator`** also *is an* **`Object`**, because all classes in C# have **`Object`** as either a direct or indirect base class. Thus, all classes in C# are connected via a hierarchical relationship in which they share the eight methods defined by class `Object`. We discuss some of these methods inherited from `Object` throughout the text.

Another inheritance hierarchy is the `Shape` hierarchy of Fig. 9.3. To specify that class `TwoDimensionalShape` is derived from (or inherits from) class `Shape`, class `TwoDimensionalShape` could be defined in C# as follows:

```
class TwoDimensionalShape : Shape
```

In Chapter 8, we briefly discussed *has-a* relationships, in which classes have as members references to objects of other classes. Such relationships create classes by *composition* of existing classes. For example, given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it is improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`. However, it is appropriate to say that an `Employee` *has a* `BirthDate` and that an `Employee` *has a* `TelephoneNumber`.

With inheritance, `private` members of a base class are not accessible directly from that class's derived classes, but these `private` base-class members are still inherited. All other base-class members retain their original member access when they become members of the derived class (e.g., `public` members of the base class become `public` members of the derived class, and, as we will soon see, `protected` members of the base class become `protected` members of the derived class). Through these inherited base-class members, the derived class can manipulate `private` members of the base class (if these inherited members provide such functionality in the base class).

It is possible to treat base-class objects and derived-class objects similarly; their commonalities are expressed in the member variables, properties and methods of the base class. Objects of all classes derived from a common base class can be treated as objects of that base class. In Chapter 10, Object-Oriented Programming: Polymorphism we consider many examples that take advantage of this relationship.

Software Engineering Observation 9.3

Constructors never are inherited—they are specific to the class in which they are defined.

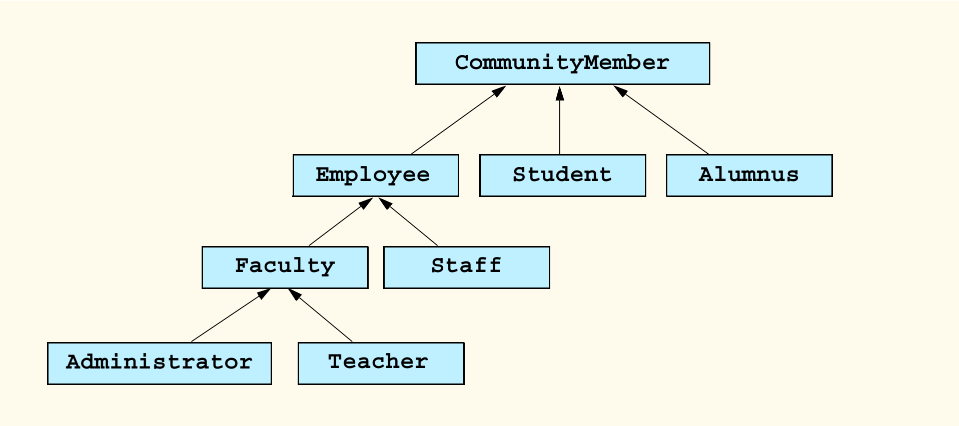


Fig. 9.2 Inheritance hierarchy for university `CommunityMember`s.

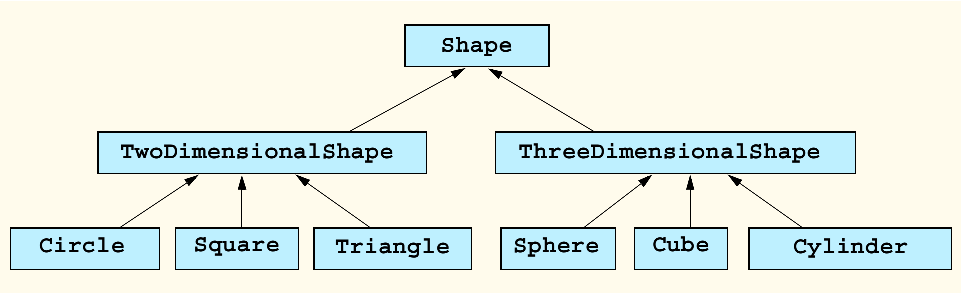


Fig. 9.3 Portion of a `Shape` class hierarchy.

9.3 `protected` and `internal` Members

Chapter 8 discussed `public` and `private` member access modifiers. A base class's `public` members are accessible anywhere that the program has a reference to an object of that base class or one of its derived classes. A base class's `private` members are accessible only within the body of that base class. In this section, we introduce two additional member access modifiers, ***`protected`*** and ***`internal`***.

Using `protected` access offers an intermediate level of protection between `public` and `private` access. A base class's `protected` members can be accessed only in that base class or in any classes derived from that base class.

Another intermediate level of access is known as `internal` access. A base class's `internal` members can be accessed only by objects declared in the same assembly. Note that an `internal` member is accessible in any part of the assembly in which that `internal` member is declared.

Derived-class methods normally can refer to `public`, `protected` and `internal` members of the base class simply by using the member names. When a derived-class method overrides a base-class member, the base-class member can be accessed from the derived class by preceding the base-class member name with keyword ***`base`***, followed by the dot operator (`.`). We discuss keyword `base` in Section 9.4.

9.4 Relationship between Base Classes and Derived Classes

In this section, we use a point-circle hierarchy[1] to discuss the relationship between a base class and a derived class. We divide our discussion of the point-circle relationship into several parts. First, we create class `Point`, which directly inherits from class `System.Object` and contains as `private` data an *x-y* coordinate pair. Then, we create class `Circle`, which also directly inherits from class `System.Object` and contains as `private` data an *x-y* coordinate pair (representing the location of the center of the circle) and a radius. We do not use inheritance to create class `Circle`; rather, we construct the class by writing every line of code the class requires. Next, we create a separate `Circle2` class, which directly inherits from class `Point` (i.e., class `Circle2` “is a” `Point` but also contains a radius) and attempts to use the `Point` `private` members—this results in compilation errors, because the derived class does not have access to the base-class’s `private` data. We then show that if `Point`’s data is declared as `protected`, a `Circle3` class that inherits from class `Point` can access that data. Both the inherited and non-inherited `Circle` classes contain identical functionality, but we show how the inherited `Circle3` class is easier to create and manage. After discussing the merits of using `protected` data, we set the `Point` data back to `private` (to enforce good software engineering), then show how a separate `Circle4` class (which also inherits from class `Point`) can use `Point` methods to manipulate `Point`’s `private` data.

---

[1] The point-circle relationship may seem unnatural when we discuss it in the context of a circle "is a" point. This example teaches what is sometimes called *structural inheritance*; the example focuses on the "mechanics" of inheritance and how a base class and a derived class relate to one another. In Chapter 10, we present more natural inheritance examples.

Let us first examine the `Point` (Fig. 9.4) class definition. The `public` services of class `Point` include two `Point` constructors (lines 13–24), properties `X` and `Y` (lines 27–54) and method `ToString` (lines 57–60). The instance variables `x` and `y` of `Point` are specified as `private` (line 10), so objects of other classes cannot access `x` and `y` directly. Technically, even if `Point`’s variables `x` and `y` were made `public`, `Point` can never maintain an inconsistent state, because the *x-y* coordinate plane is infinite in both directions, so `x` and `y` can hold any `int` value. In general, however, declaring data as `private`, while providing non-`private` properties to manipulate and perform validation checking on that data, enforces good software engineering.

We mentioned in Section 9.2 that constructors are not inherited. Therefore, Class `Point` does not inherit class `Object`’s constructor. However, class `Point`’s constructors (lines 13–24) call class `Object`’s constructor implicitly. In fact, the first task of any derived-class constructor is to call its direct base class’s constructor, either implicitly or explicitly. (The syntax for calling a base-class constructor is discussed later in this section.) If the code does not include an explicit call to the base-class constructor, an implicit call is made to the base class’s default (no-argument) constructor. The comments in lines 15 and 21 indicate where the implicit calls to the base-class `Object`’s default constructor occur.

```
 1  // Fig. 9.4: Point.cs
 2  // Point class represents an x-y coordinate pair.
 3
 4  using System;
 5
 6  // Point class definition implicitly inherits from Object
 7  public class Point
 8  {
 9     // point coordinates
10     private int x, y;
11
12     // default (no-argument) constructor
13     public Point()
14     {
15        // implicit call to Object constructor occurs here
16     }
17
18     // constructor
19     public Point( int xValue, int yValue )
20     {
```

Fig. 9.4 `Point` class represents an *x-y* coordinate pair. (Part 1 of 2.)

```
21        // implicit call to Object constructor occurs here
22        X = xValue;
23        Y = yValue;
24     }
25
26     // property X
27     public int X
28     {
29        get
30        {
31           return x;
32        }
33
34        set
35        {
36           x = value; // no need for validation
37        }
38
39     } // end property X
40
41     // property Y
42     public int Y
43     {
44        get
45        {
46           return y;
47        }
48
49        set
50        {
51           y = value; // no need for validation
52        }
53
54     } // end property Y
55
56     // return string representation of Point
57     public override string ToString()
58     {
59        return "[" + x + ", " + y + "]";
60     }
61
62  } // end class Point
```

Fig. 9.4 `Point` class represents an *x-y* coordinate pair. (Part 2 of 2.)

Note that method **`ToString`** (lines 57–60) contains the keyword ***`override`*** in its declaration. Every class in C# (such as class **`Point`**) inherits either directly or indirectly from class **`System.Object`**, which is the root of the class hierarchy. As we mentioned previously, this means that every class inherits the eight methods defined by class **`Object`**. One of these methods is ***`ToString`***, which returns a **`string`** containing the object's type preceded by its namespace—this method obtains an object's **`string`** representation and sometimes is called implicitly by the program (such as when an object is concatenated to a **`string`**). Method **`ToString`** of class **`Point`** *overrides* the original **`ToString`** from class **`Object`**—when invoked, method **`ToString`** of class **`Point`** returns a **`string`** containing an ordered pair of the values **`x`** and **`y`** (line 59), instead of returning a **`string`** containing the object's class and namespace. To override a base-class method definition, a derived class must specify that the derived-class method overrides the base-class method with keyword **`override`** in the method header.

Software Engineering Observation 9.4

*The C# compiler sets the base class of a derived class to **`Object`** when the program does not specify a base class explicitly.*

In C#, a base-class method must be declared ***`virtual`*** if that method is to be overridden in a derived class. Method **`ToString`** of class **`Object`** is, in fact, declared **`virtual`**, which enables derived class **`Point`** to override this method. To view the method header for **`ToString`**, select **Help > Index...**, and enter **`Object.ToString method`** (filtered by **.Net Framework SDK**) in the search text box. The page displayed contains a description of method **`ToString`**, which includes the following header:

```
public virtual string ToString();
```

Keyword **`virtual`** allows programmers to specify those methods that a derived class can override—a method that has not been declared **`virtual`** cannot be overridden. We use this later in this section to enable certain methods in our base classes to be overridden.

Common Programming Error 9.1

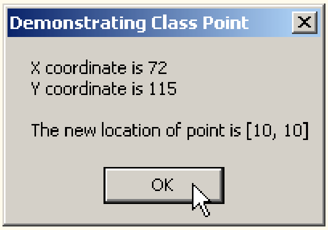
*A derived class attempting to override (using keyword **`override`**) a method that has not been declared **`virtual`** is a syntax error.*

Class **`PointTest`** (Fig. 9.5) tests class **`Point`**. Line 14 instantiates an object of class **`Point`** and assigns **`72`** as the *x*-coordinate value and **`115`** as the *y*-coordinate value. Lines 17–18 use properties **`X`** and **`Y`** to retrieve these values, then append the values to **`string output`**. Lines 20–21 change the values of properties **`X`** and **`Y`** (implicitly invoking their **`set`** accessors), and line 24 calls **`Point`**'s **`ToString`** method implicitly to obtain the **`Point`**'s **`string`** representation.

```
 1  // Fig. 9.5: PointTest.cs
 2  // Testing class Point.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // PointTest class definition
 8  class PointTest
 9  {
10     // main entry point for application
11     static void Main( string[] args )
12     {
13        // instantiate Point object
14        Point point = new Point( 72, 115 );
15
```

Fig. 9.5 **`PointTest`** class demonstrates class **`Point`** functionality. (Part 1 of 2.)

```
      // display point coordinates via X and Y properties
      string output = "X coordinate is " + point.X +
         "\n" + "Y coordinate is " + point.Y;

      point.X = 10; // set x-coordinate via X property
      point.Y = 10; // set y-coordinate via Y property

      // display new point value
      output += "\n\nThe new location of point is " + point;

      MessageBox.Show( output, "Demonstrating Class Point" );

   } // end method Main

} // end class PointTest
```



```
X coordinate is 72
Y coordinate is 115

The new location of point is [10, 10]
```

Fig. 9.5 `PointTest` class demonstrates class `Point` functionality. (Part 2 of 2.)

We now discuss the second part of our introduction to inheritance by creating and testing (a completely new) class **`Circle`** (Fig. 9.6), which directly inherits from class **`System.Object`** and represents an *x-y* coordinate pair (representing the center of the circle) and a radius. Lines 9–10 declare the instance variables **`x`**, **`y`** and **`radius`** as **`private`** data. The **`public`** services of class **`Circle`** include two **`Circle`** constructors (lines 13–25), properties **`X`**, **`Y`** and **`Radius`** (lines 28–71), methods **`Diameter`** (lines 74–77), **`Circumference`** (lines 80–83), **`Area`** (lines 86–89) and **`ToString`** (lines 92–96). These properties and methods encapsulate all necessary features (i.e., the "analytic geometry") of a circle; in the next section, we show how this encapsulation enables us to reuse and extend this class.

```
 1  // Fig. 9.6: Circle.cs
 2  // Circle class contains x-y coordinate pair and radius.
 3
 4  using System;
 5
 6  // Circle class definition implicitly inherits from Object
 7  public class Circle
 8  {
 9     private int x, y;      // coordinates of Circle's center
10     private double radius; // Circle's radius
11
```

Fig. 9.6 `Circle` class contains an *x-y* coordinate and a radius. (Part 1 of 3.)

```
   // default constructor
   public Circle()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Circle( int xValue, int yValue, double radiusValue )
   {
      // implicit call to Object constructor occurs here
      x = xValue;
      y = yValue;
      Radius = radiusValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

      set
      {
         x = value;  // no need for validation
      }

   } // end property X

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value;  // no need for validation
      }

   } // end property Y

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

```

Fig. 9.6 `Circle` class contains an *x*-*y* coordinate and a radius. (Part 2 of 3.)

```
      set
      {
         if ( value >= 0 )    // validation needed
            radius = value;
      }

   } // end property Radius

   // calculate Circle diameter
   public double Diameter()
   {
      return radius * 2;
   }

   // calculate Circle circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle area
   public double Area()
   {
      return Math.PI * Math.Pow( radius, 2 );
   }

   // return string representation of Circle
   public override string ToString()
   {
      return "Center = [" + x + ", " + y + "]" +
         "; Radius = " + radius;
   }

} // end class Circle
```

Fig. 9.6 **`Circle`** class contains an *x-y* coordinate and a radius. (Part 3 of 3.)

Class **`CircleTest`** (Fig. 9.7) tests class **`Circle`**. Line 14 instantiates an object of class **`Circle`**, assigning **`37`** as the *x*-coordinate value, **`43`** as the *y*-coordinate value and **`2.5`** as the radius value. Lines 17–19 use properties **`X`**, **`Y`** and **`Radius`** to retrieve these values, then concatenate the values to **`string output`**. Lines 22–24 use **`Circle`**'s **`X`**, **`Y`** and **`Radius`** properties to change the *x-y* coordinates and the radius, respectively. Property **`Radius`** ensures that member variable **`radius`** cannot be assigned a negative value. Line 28 calls **`Circle`**'s **`ToString`** method implicitly to obtain the **`Circle`**'s **`string`** representation, and lines 32–40 call **`Circle`**'s **`Diameter`**, **`Circumference`** and **`Area`** methods.

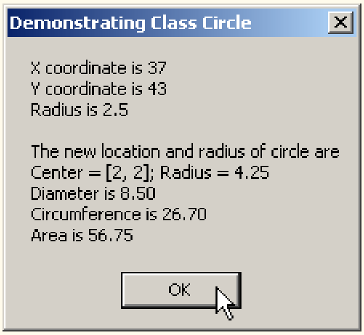
After writing all the code for class **`Circle`** (Fig. 9.6), we note that a major portion of the code in this class is similar, if not identical, to much of the code in class **`Point`**. For example, the declaration in **`Circle`** of **`private`** variables **`x`** and **`y`** and properties **`X`** and **`Y`** are identical to those of class **`Point`**. In addition, the class **`Circle`** constructors and method **`ToString`** are almost identical to those of class **`Point`**, except that they also supply **`radius`** information. The only other additions to class **`Circle`** are **`private`** member variable **`radius`**, property **`Radius`** and methods **`Diameter`**, **`Circumference`** and **`Area`**.

It appears that we literally copied code from class **`Point`**, pasted this code in the code from class **`Circle`**, then modified class **`Circle`** to include a radius. This "copy-and-paste" approach is often error-prone and time-consuming. Worse yet, it can result in many physical copies of the code existing throughout a system, creating a code-maintenance "nightmare." Is there a way to "absorb" the attributes and behaviors of one class in a way that makes them part of other classes without duplicating code?

```
// Fig. 9.7: CircleTest.cs
// Testing class Circle.

using System;
using System.Windows.Forms;

// CircleTest class definition
class CircleTest
{
   // main entry point for application.
   static void Main( string[] args )
   {
      // instantiate Circle
      Circle circle = new Circle( 37, 43, 2.5 );

      // get Circle's initial x-y coordinates and radius
      string output = "X coordinate is " + circle.X +
         "\nY coordinate is " + circle.Y + "\nRadius is " +
         circle.Radius;

      // set Circle's x-y coordinates and radius to new values
      circle.X = 2;
      circle.Y = 2;
      circle.Radius = 4.25;

      // display Circle's string representation
      output += "\n\nThe new location and radius of " +
         "circle are \n" + circle + "\n";

      // display Circle's diameter
      output += "Diameter is " +
         String.Format( "{0:F}", circle.Diameter() ) + "\n";

      // display Circle's circumference
      output += "Circumference is " +
         String.Format( "{0:F}", circle.Circumference() ) + "\n";

      // display Circle's area
      output += "Area is " +
         String.Format( "{0:F}", circle.Area() );

      MessageBox.Show( output, "Demonstrating Class Circle" );

   } // end method Main
```

Fig. 9.7 **`CircleTest`** demonstrates class **`Circle`** functionality. (Part 1 of 2.)

```

} // end class CircleTest
```



```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.75
```

Fig. 9.7 **`CircleTest`** demonstrates class **`Circle`** functionality. (Part 2 of 2.)

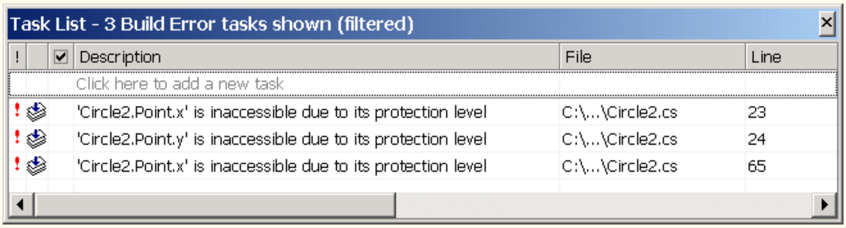
In the next examples we answer that question, we use a more elegant class construction approach emphasizing the benefits of inheritance. Now, we create and test a class **`Circle2`** (Fig. 9.8) that inherits variables **`x`** and **`y`** and properties **`X`** and **`Y`** from class **`Point`** (Fig. 9.4). This class **`Circle2`** "is a" **`Point`** (because inheritance absorbs the capabilities of class **`Point`**), but also contains **`radius`** (line 9). The colon (**`:`**) symbol in the class declaration (line 7) indicates inheritance. As a derived class, **`Circle2`** inherits all the members of class **`Point`**, except for the constructors. Thus, the **`public`** services to **`Circle2`** include the two **`Circle2`** constructors (lines 12–24); the **`public`** methods inherited from class **`Point`**; property **`Radius`** (lines 27–40); and the **`Circle2`** methods **`Diameter`**, **`Circumference`**, **`Area`** and **`ToString`** (lines 43–65). We declare method **`Area`** as **`virtual`**, so that derived classes (such as class **`Cylinder`**, as we will see in Section 9.5) can override this method to provide a more appropriate implementation.

```
 1  // Fig. 9.8: Circle2.cs
 2  // Circle2 class that inherits from class Point.
 3
 4  using System;
 5
 6  // Circle2 class definition inherits from Point
 7  class Circle2 : Point
 8  {
 9     private double radius; // Circle2's radius
10
11     // default constructor
12     public Circle2()
13     {
14        // implicit call to Point constructor occurs here
15     }
16
```

Fig. 9.8 **`Circle2`** class that inherits from class **`Point`**. (Part 1 of 3.)

```
   // constructor
   public Circle2( int xValue, int yValue, double radiusValue )
   {
      // implicit call to Point constructor occurs here
      x = xValue;
      y = yValue;
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         if ( value >= 0 )
            radius = value;
      }

   } // end property Radius

   // calculate Circle diameter
   public double Diameter()
   {
      return radius * 2;
   }

   // calculate Circle circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle area
   public virtual double area()
   {
      return Math.PI * Math.Pow( radius, 2 );
   }

   // return string representation Circle
   public override string ToString()
   {
      return "Center = [" + x + ", " + y + "]" +
         "; Radius = " + radius;
   }

} // end class Circle2
```

Fig. 9.8 **`Circle2`** class that inherits from class **`Point`**. (Part 2 of 3.)



Task List - 3 Build Error tasks shown (filtered)

!	☑	Description	File	Line
		Click here to add a new task		
!		'Circle2.Point.x' is inaccessible due to its protection level	C:\...\Circle2.cs	23
!		'Circle2.Point.y' is inaccessible due to its protection level	C:\...\Circle2.cs	24
!		'Circle2.Point.x' is inaccessible due to its protection level	C:\...\Circle2.cs	65

Fig. 9.8 `Circle2` class that inherits from class `Point`. (Part 3 of 3.)

Lines 14 and 20 in the `Circle2` constructors (lines 12–24) invoke the default `Point` constructor implicitly to initialize the base-class portion (variables `x` and `y`, inherited from class `Point`) of a `Circle2` object to `0`. However, because the parameterized constructor (lines 18–24) should set the *x-y* coordinate to a specific value, lines 21–22 attempt to assign argument values to `x` and `y` directly. Even though lines 21–22 attempt to set `x` and `y` values explicitly, line 20 first calls the `Point` default constructor to initialize these variables to their default values. The compiler generates syntax errors for lines 21 and 22 (and line 63, where `Circle2`'s method `ToString` attempts to use the values of `x` and `y` directly), because the derived class `Circle2` is not allowed to access the base class `Point`'s `private` members `x` and `y`. C# rigidly enforces restriction on accessing `private` data members, so that even a derived class (i.e., which is closely related to its base class) cannot access base-class `private` data.

To enable class `Circle2` to access `Point` member variables `x` and `y` directly, we can declare those variables as `protected`. As we discussed in Section 9.3, a base class's `protected` members can be accessed only in that base class or in any classes derived from that base class. Class `Point2` (Fig. 9.9) modifies class `Point` (Fig. 9.4) to declare variables `x` and `y` as `protected` (line 10) instead of `private`.

```
 1  // Fig. 9.9: Point2.cs
 2  // Point2 class contains an x-y coordinate pair as protected data.
 3
 4  using System;
 5
 6  // Point2 class definition implicitly inherits from Object
 7  public class Point2
 8  {
 9     // point coordinate
10     protected int x, y;
11
12     // default constructor
13     public Point2()
14     {
15        // implicit call to Object constructor occurs here
16     }
17
```

Fig. 9.9 `Point2` class represents an *x-y* coordinate pair as `protected` data. (Part 1 of 2.)

```
   // constructor
   public Point2( int xValue, int yValue )
   {
      // implicit call to Object constructor occurs here
      X = xValue;
      Y = yValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

      set
      {
         x = value; // no need for validation
      }

   } // end property X

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value; // no need for validation
      }

   } // end property Y

   // return string representation of Point2
   public override string ToString()
   {
      return "[" + x + ", " + y + "]";
   }

} // end class Point2
```

Fig. 9.9 `Point2` class represents an *x-y* coordinate pair as `protected` data. (Part 2 of 2.)

Class **`Circle3`** (Fig. 9.10) modifies class **`Circle2`** (Fig. 9.8) to inherit from class **`Point2`** rather than inheriting from class **`Point`**. Because class **`Circle3`** is a class derived from class **`Point2`**, class **`Circle3`** can access class **`Point2`**'s **`protected`** member variables **`x`** and **`y`** directly, and the compiler does not generate errors when compiling Fig. 9.10. This shows the special privileges that a derived class is granted to access

`protected` base-class data members. A derived class also can access **`protected`** methods in any of that derived class's base classes.

```
// Fig. 9.10: Circle3.cs
// Circle2 class that inherits from class Point2.

using System;

// Circle3 class definition inherits from Point2
public class Circle3 : Point2
{
   private double radius; // Circle's radius

   // default constructor
   public Circle3()
   {
      // implicit call to Point constructor occurs here
   }

   // constructor
   public Circle3(
      int xValue, int yValue, double radiusValue )
   {
      // implicit call to Point constructor occurs here
      x = xValue;
      y = yValue;
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         if ( value >= 0 )
            radius = value;
      }

   } // end property Radius

   // calculate Circle diameter
   public double Diameter()
   {
      return radius * 2;
   }

```

Fig. 9.10 `Circle3` class that inherits from class `Point2`. (Part 1 of 2.)

```
   // calculate circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle area
   public virtual double Area()
   {
      return Math.PI * Math.Pow( radius, 2 );
   }

   // return string representation of Circle3
   public override string ToString()
   {
      return "Center = [" + x + ", " + y + "]" +
         "; Radius = " + radius;
   }

} // end class Circle3
```

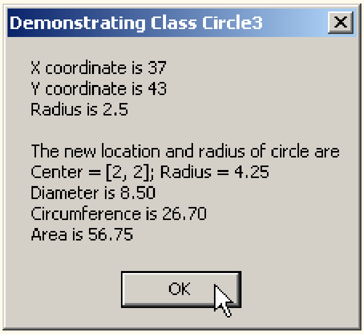
Fig. 9.10 **`Circle3`** class that inherits from class **`Point2`**. (Part 2 of 2.)

Class **`CircleTest3`** (Fig. 9.11) performs identical tests on class **`Circle3`** as class **`CircleTest`** (Fig. 9.7) performed on class **`Circle`** (Fig. 9.6). Note that the outputs of the two programs are identical. We created class **`Circle`** without using inheritance and created class **`Circle3`** using inheritance; however, both classes provide the same functionality. However, observe that the code listing for class **`Circle3`**, which is 68 lines, is considerably shorter than the code listing for class **`Circle`**, which is 98 lines, because class **`Circle3`** absorbs part of its functionality from **`Point2`**, whereas class **`Circle`** does not. Also, there is now only one copy of the point functionality.

```
/ Fig. 9.11: CircleTest3.cs
// Testing class Circle3.

using System;
using System.Windows.Forms;

// CircleTest3 class definition
class CircleTest3
{
   // main entry point for application
   static void Main( string[] args )
   {
      // instantiate Circle3
      Circle3 circle = new Circle3( 37, 43, 2.5 );

      // get Circle3's initial x-y coordinates and radius
      string output = "X coordinate is " + circle.X + "\n" +
         "Y coordinate is " + circle.Y + "\nRadius is " +
         circle.Radius;
```

Fig. 9.11 **`CircleTest3`** demonstrates class **`Circle3`** functionality. (Part 1 of 2.)

```

      // set Circle3's x-y coordinates and radius to new values
      circle.X = 2;
      circle.Y = 2;
      circle.Radius = 4.25;

      // display Circle3's string representation
      output += "\n\n" +
         "The new location and radius of circle are " +
         "\n" + circle + "\n";

      // display Circle3's Diameter
      output += "Diameter is " +
         String.Format( "{0:F}", circle.Diameter() ) + "\n";

      // display Circle3's Circumference
      output += "Circumference is " +
         String.Format( "{0:F}", circle.Circumference() ) + "\n";

      // display Circle3's Area
      output += "Area is " +
         String.Format( "{0:F}", circle.Area() );

      MessageBox.Show( output, "Demonstrating Class Circle3" );

   } // end method Main

} // end class CircleTest3
```



```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.75
```

Fig. 9.11 `CircleTest3` demonstrates class `Circle3` functionality. (Part 2 of 2.)

In the previous example, we declared the base-class instance variables as **`protected`**, so that a derived class could modify their values directly. The use of **`protected`** variables allows for a slight increase in performance, because we avoid incurring the overhead of a method call to a property's **`set`** or **`get`** accessor. However, in most C# applications, in which user interaction comprises a large part of the execution time, the optimization offered through the use of **`protected`** variables is negligible.

Using **`protected`** instance variables creates two major problems. First, the derived-class object does not have to use a property to set the value of the base-class's **`protected`** data. Therefore, a derived-class object can easily assign an illegal value to the **`protected`** data, thus leaving the object in an inconsistent state. For example, if we were to declare `Circle3`'s variable `radius` as `protected`, a derived-class object (e.g., `Cylinder`), could then assign a negative value to `radius`. The second problem with using `protected` data is that derived-class methods are more likely to be written to depend on base-class implementation. In practice, derived classes should depend only on the base-class services (i.e., non-`private` methods and properties) and not on base-class implementation. With `protected` data in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class. For example, if for some reason we were to change the names of variables `x` and `y` to `xCoordinate` and `yCoordinate`, then we would have to do so for all occurrences in which a derived class references these base-class variables directly. In such a case, the software is said to be *fragile* or *brittle*. The programmer should be able to change the base-class implementation freely, while still providing the same services to derived classes. (Of course, if the base class services change, we must reimplement our derived classes, but good object-oriented design attempts to prevent this.)

Software Engineering Observation 9.5

The most appropriate time to use the `protected` *access modifier is when a base class should provide a service only to its derived classes (i.e., the base class should not provide the service to other clients).*

Software Engineering Observation 9.6

Declaring base-class instance variables `private` *(as opposed to declaring them* `protected`*) enables programmers to change base-class implementation without having to change derived-class implementation.*

Testing and Debugging Tip 9.1

When possible, avoid including `protected` *data in a base class. Rather, include non-*`private` *properties and methods that access* `private` *data, ensuring that the object maintains a consistent state.*

We reexamine our point-circle hierarchy example once more; this time, attempting to use the best software engineering. We use `Point3` (Fig. 9.12), which declares variables `x` and `y` as `private` and uses properties in method `ToString` to access these values. We show how derived class `Circle4` (Fig. 9.13) can invoke non-`private` base-class methods and properties to manipulate these variables.

```
// Fig. 9.12: Point3.cs
// Point3 class represents an x-y coordinate pair.

using System;

// Point3 class definition implicitly inherits from Object
public class Point3
{
   // point coordinate
   private int x, y;

```

Fig. 9.12 `Point3` class uses properties to manipulate its `private` data. (Part 1 of 2.)

```
   // default constructor
   public Point3()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point3( int xValue, int yValue )
   {
      // implicit call to Object constructor occurs here
      X = xValue;   // use property X
      Y = yValue;   // use property Y
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

      set
      {
         x = value; // no need for validation
      }

   } // end property X

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value; // no need for validation
      }

   } // end property Y

   // return string representation of Point3
   public override string ToString()
   {
      return "[" + X + ", " + Y + "]";
   }

} // end class Point3
```

Fig. 9.12 `Point3` class uses properties to manipulate its `private` data. (Part 2 of 2.)

```
// Fig. 9.13: Circle4.cs
// Circle4 class that inherits from class Point3.

using System;

// Circle4 class definition inherits from Point3
public class Circle4 : Point3
{
   private double radius;

   // default constructor
   public Circle4()
   {
      // implicit call to Point constructor occurs here
   }

   // constructor
   public Circle4( int xValue, int yValue, double radiusValue )
      : base( xValue, yValue )
   {
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         if ( value >= 0 )    // validation needed
            radius = value;
      }

   } // end property Radius

   // calculate Circle diameter
   public double Diameter()
   {
      return Radius * 2;      // use property Radius
   }

   // calculate Circle circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

```

Fig. 9.13 **`Circle4`** class that inherits from class **`Point3`**, which does not provide **`protected`** data (Part 1 of 2.).

```
52      // calculate Circle area
53      public virtual double Area()
54      {
55         return Math.PI * Math.Pow( Radius, 2 );  // use property
56      }
57
58      // return string representation of Circle4
59      public override string ToString()
60      {
61         // use base reference to return Point string representation
62         return "Center= " + base.ToString() +
63            "; Radius = " + Radius;  // use property Radius
64      }
65
66   } // end class Circle4
```

Fig. 9.13 `Circle4` class that inherits from class `Point3`, which does not provide `protected` data (Part 2 of 2.).

Software Engineering Observation 9.7

When possible, use properties to alter and obtain the values of member variables, even if those values can be modified directly. A property's **`set`** *accessor can prevent attempts to assign an inappropriate value to that the value, and a property's* **`get`** *accessor can help control the presentation of the data to clients.*

Performance Tip 9.1

Using a property to access a variable's value is slightly slower than accessing the data directly. However, attempting to optimize programs by referencing data directly often is unnecessary, because the compiler optimizes the programs implicitly. [Today's so-called "optimizing compilers" are carefully designed to perform many optimizations implicitly, even if the programmer does not write what appears to be the most optimal code. A good rule is, "Do not second-guess the compiler."

For the purpose of this example, to demonstrate both explicit and implicit calls to base-class constructors, we include a second constructor that calls the base-class constructor explicitly. Lines 18–22 declare the **`Circle4`** constructor that invokes the second **`Point3`** constructor explicitly (line 19) using the *base-class constructor-call syntax* (i.e., reference ***`base`*** followed by a set of parentheses containing the arguments to the base-class constructor). In this case, **`xValue`** and **`yValue`** are passed to initialize the **`private`** base-class members **`x`** and **`y`**. The colon symbol (**`:`**) followed by the **`base`** keyword accesses the base-class version of that method explicitly (line 19). By making this call, we can initialize **`x`** and **`y`** in the base class to specific values, rather than to **`0`**.

Common Programming Error 9.2

It is a syntax error if a derived class uses **`base`** *to call its base-class constructor with arguments that do not match exactly the number and types of parameters specified in one of the base-class constructor definitions.*

Class **`Circle4`**'s **`ToString`** method (line 59–64) overrides class **`Point3`**'s **`ToString`** method (lines 57–60 of Fig. 9.12). As we discussed earlier, overriding this method is possible, because method **`ToString`** of class **`System.Object`** (class

Point3's base class) is declared **virtual**. Method **ToString** of class **Circle4** displays the **private** instance variables **x** and **y** of class **Point3** by calling the base class's **ToString** method (in this case, **Point3**'s **ToString** method). The call is made in line 62 via the expression **base.ToString()** and causes the values of **x** and **y** to become part of the **Circle4**'s **string** representation. Using this approach is a good software engineering practice: Recall that *Software Engineering Observation* 8.11 stated that, if an object's method performs the actions needed by another object, call that method rather than duplicating its code body. Duplicate code creates code-maintenance problems. By having **Circle4**'s **ToString** method use the formatting provided by **Point3**'s **ToString** method, we avoid duplicating code. Also, **Point3**'s **ToString** method performs part of the task of **Circle4**'s **ToString** method, so we call **Point3**'s **ToString** method from class **Circle4** with the expression **base.ToString()**.

Common Programming Error 9.3

When a base-class method is overridden in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the ***base*** *reference when referencing the base class's method causes infinite recursion, because the derived-class method would then call itself.*

Common Programming Error 9.4

The use of "chained" ***base*** *references to refer to a member (a method, property or variable) several levels up the hierarchy (as in* ***base.base.mX****) is a syntax error.*

Software Engineering Observation 9.8

A redefinition in a derived class of a base-class method that uses a different signature than that of the base-class method is method overloading rather than method overriding.

Software Engineering Observation 9.9

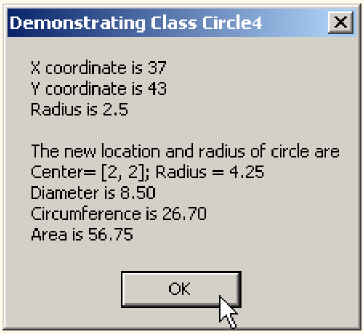
Although method ***ToString*** *certainly could be overridden to perform arbitrary actions, the general understanding in the C# .NET community is that method* ***ToString*** *should be overridden to obtain an object's* ***string*** *representation.*

Class **CircleTest4** (Fig. 9.14) performs identical manipulations on class **Circle4** as did classes **CircleTest** (Fig. 9.7) and **CircleTest3** (Fig. 9.11). Note that the outputs of all three modules are identical. Therefore, although each "circle" class appears to behave identically, class **Circle4** is the most properly engineered. Using inheritance, we have efficiently and effectively constructed a well-engineered class.

```
// Fig. 9.14: CircleTest4.cs
// Testing class Circle4.

using System;
using System.Windows.Forms;

// CircleTest4 class definition
class CircleTest4
{
```

Fig. 9.14 **CircleTest4** demonstrates class **Circle4** functionality. (Part 1 of 2.)

```
   // main entry point for application
   static void Main( string[] args )
   {
      // instantiate Circle4
      Circle4 circle = new Circle4( 37, 43, 2.5 );

      // get Circle4's initial x-y coordinates and radius
      string output = "X coordinate is " + circle.X + "\n" +
         "Y coordinate is " + circle.Y + "\n" +
         "Radius is " + circle.Radius;

      // set Circle4's x-y coordinates and radius to new values
      circle.X = 2;
      circle.Y = 2;
      circle.Radius = 4.25;

      // display Circle4's string representation
      output += "\n\n" +
         "The new location and radius of circle are " +
         "\n" + circle + "\n";

      // display Circle4's Diameter
      output += "Diameter is " +
         String.Format( "{0:F}", circle.Diameter() ) + "\n";

      // display Circle4's Circumference
      output += "Circumference is " +
         String.Format( "{0:F}", circle.Circumference() ) + "\n";

      // display Circle4's Area
      output += "Area is " +
         String.Format( "{0:F}", circle.Area() );

      MessageBox.Show( output, "Demonstrating Class Circle4" );

   } // end method Main

} // end class CircleTest4
```



```
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center= [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.75
```

Fig. 9.14 `CircleTest4` demonstrates class `Circle4` functionality. (Part 2 of 2.)

9.5 Case Study: Three-Level Inheritance Hierarchy

Let us consider a more substantial inheritance example involving a three-level point-circle-cylinder hierarchy. In Section 9.4, we developed classes `Point3` (Fig. 9.12) and `Circle4` (Fig. 9.13). Now, we present an example in which we derive class `Cylinder` from class `Circle4`.

The first class that we use in our case study is class `Point3` (Fig. 9.12). We declared `Point3`'s instance variables as `private`. Class `Point3` also contains properties `X` and `Y` for accessing `x` and `y` and method `ToString` (which `Point3` overrides from class `Object`) for obtaining a `string` representation of the *x*-*y* coordinate pair.

We also created class `Circle4` (Fig. 9.13), which inherits from class `Point3`. Class `Circle4` contains the `Point3` functionality, in addition to providing property `Radius`, which ensures that the `radius` member variable cannot hold a negative value, and methods `Diameter`, `Circumference`, `Area` and `ToString`. Recall that method `Area` was declared `virtual` (line 53). As we discussed in Section 9.4, this keyword enables derived classes to override a base-class method. Derived classes of class `Circle4` (such as class `Cylinder`, which we introduce momentarily) can override these methods and provide specific implementations. A circle has an area that is calculated by the formula, πr^2, in which r represents the circle's radius. However, a cylinder has a surface area that is calculated by the formula, $(2\pi r^2) + (2\pi rh)$, in which r represents the cylinder's radius and h represents the cylinder's height. Therefore, class `Cylinder` must override method `Area` to include this calculation, so we declared class `Circle4`'s method `Area` as `virtual`.

Figure 9.15 presents class `Cylinder`, which inherits from class `Circle4` (line 7). Class `Cylinder`'s `public` services include the inherited `Circle4` methods `Diameter`, `Circumference`, `Area` and `ToString`; the inherited `Circle4` property `Radius`; the indirectly inherited `Point3` properties `X` and `Y`; the `Cylinder` constructor, property `Height` and method `Volume`. Method `Area` (lines 41–44) overrides method `Area` of class `Circle4`. Note that, if class `Cylinder` were to attempt to override `Circle4`'s methods `Diameter` and `Circumference`, syntax errors would occur, because class `Circle4` did not declare these methods `virtual`. Method `ToString` (lines 53–56) overrides method `ToString` of class `Circle4` to obtain a `string` representation for the cylinder. Class `Cylinder` also includes method `Volume` (lines 47–50) to calculate the cylinder's volume. Because we do not declare method `Volume` as `virtual`, no derived class of class `Cylinder` can override this method.

```
 1  // Fig. 9.15: Cylinder.cs
 2  // Cylinder class inherits from class Circle4.
 3
 4  using System;
 5
 6  // Cylinder class definition inherits from Circle4
 7  public class Cylinder : Circle4
 8  {
 9     private double height;
10
```

Fig. 9.15 `Cylinder` class inherits from class `Circle4` and overrides method `Area`. (Part 1 of 2.)

```
   // default constructor
   public Cylinder()
   {
      // implicit call to Circle4 constructor occurs here
   }

   // four-argument constructor
   public Cylinder( int xValue, int yValue, double radiusValue,
      double heightValue ) : base( xValue, yValue, radiusValue )
   {
      Height = heightValue; // set Cylinder height
   }

   // property Height
   public double Height
   {
      get
      {
         return height;
      }

      set
      {
         if ( value >= 0 ) // validate height
           height = value;
      }

   } // end property Height

   // override Circle4 method Area to calculate Cylinder area
   public override double Area()
   {
      return 2 * base.Area() + base.Circumference() * Height;
   }

   // calculate Cylinder volume
   public double Volume()
   {
      return base.Area() * Height;
   }

   // convert Cylinder to string
   public override string ToString()
   {
      return base.ToString() + "; Height = " + Height;
   }

} // end class Cylinder
```

Fig. 9.15 **`Cylinder`** class inherits from class **`Circle4`** and overrides method **`Area`**. (Part 2 of 2.)

Figure 9.16 is a **`CylinderTest`** application that tests the **`Cylinder`** class. Line 14 instantiates an object of class **`Cylinder`**. Lines 17–19 use properties **`X`**, **`Y`**, **`Radius`** and

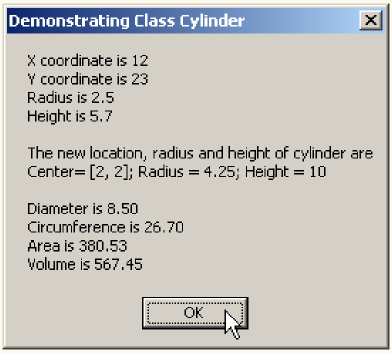
Height to obtain information about the **Cylinder** object, because **CylinderTest** cannot reference the **private** data of class **Cylinder** directly. Lines 22–25 use properties **X**, **Y**, **Radius** and **Height** to reset the **Cylinder**'s *x-y* coordinates (we assume the cylinder's *x-y* coordinates specify its position on the *x-y* plane), radius and height. Class **Cylinder** can use class **Point3**'s **X** and **Y** properties, because class **Cylinder** inherits them indirectly from class **Point3**—Class **Cylinder** inherits properties **X** and **Y** directly from class **Circle4**, which inherited them directly from class **Point3**. Line 29 invokes method **ToString** implicitly to obtain the **string** representation of the **Cylinder** object. Lines 33–37 invoke methods **Diameter** and **Circumference** of the **Cylinder** object—because class **Cylinder** inherits these methods from class **Circle4** and cannot override them, these methods, exactly as listed in **Circle4**, are invoked. Lines 41–45 invoke methods **Area** and **Volume**.

Using the point-circle-cylinder example, we have shown the use and benefits of inheritance. We were able to develop classes **Circle4** and **Cylinder** using inheritance much faster than if we had developed these classes "from scratch." Inheritance avoids duplicating code and the associated code-maintenance problems.

```
 1  // Fig. 9.16: CylinderTest.cs
 2  // Tests class Cylinder.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // CylinderTest class definition
 8  class CylinderTest
 9  {
10     // main entry point for application
11     static void Main( string[] args )
12     {
13        // instantiate object of class Cylinder
14        Cylinder cylinder = new Cylinder(12, 23, 2.5, 5.7);
15
16        // properties get initial x-y coordinate, radius and height
17        string output = "X coordinate is " + cylinder.X + "\n" +
18           "Y coordinate is " + cylinder.Y + "\nRadius is " +
19           cylinder.Radius + "\n" + "Height is " + cylinder.Height;
20
21        // properties set new x-y coordinate, radius and height
22        cylinder.X = 2;
23        cylinder.Y = 2;
24        cylinder.Radius = 4.25;
25        cylinder.Height = 10;
26
27        // get new x-y coordinate and radius
28        output += "\n\nThe new location, radius and height of " +
29           "cylinder are\n" + cylinder + "\n\n";
30
31        // display Cylinder's Diameter
32        output += "Diameter is " +
33           String.Format( "{0:F}", cylinder.Diameter() ) + "\n";
```

Fig. 9.16 Testing class **Cylinder** (Part 1 of 2.).

```

      // display Cylinder's Circumference
      output += "Circumference is " +
         String.Format( "{0:F}", cylinder.Circumference() ) + "\n";

      // display Cylinder's Area
      output += "Area is " +
         String.Format( "{0:F}", cylinder.Area() ) + "\n";

      // display Cylinder's Volume
      output += "Volume is " +
         String.Format( "{0:F}", cylinder.Volume() );

      MessageBox.Show( output, "Demonstrating Class Cylinder" );

   } // end method Main

} // end class CylinderTest
```



```
X coordinate is 12
Y coordinate is 23
Radius is 2.5
Height is 5.7

The new location, radius and height of cylinder are
Center= [2, 2]; Radius = 4.25; Height = 10

Diameter is 8.50
Circumference is 26.70
Area is 380.53
Volume is 567.45
```

Fig. 9.16 Testing class **`Cylinder`** (Part 2 of 2.).

9.6 Constructors and Destructors in Derived Classes

As we explained in the previous section, instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes the base-class constructor either explicitly or implicitly. Similarly, if the base-class was derived from another class, the base-class constructor must invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in the chain is class **`Object`**'s constructor whose body actually finishes executing first—the original derived class's body finishes executing last. Each base-class constructor initializes the base-class instance variables that the derived-class object inherits. For example, consider the **`Point3`**/**`Circle4`** hierarchy from Fig. 9.12 and Fig. 9.13. When a program creates a **`Circle4`** object, one of the **`Circle4`** constructors is called. That constructor calls class **`Point3`**'s constructor, which in turn calls class **`Object`**'s constructor. When class **`Object`**'s constructor completes execution, it returns control to class **`Point3`**'s constructor, which initializes the *x*-*y* coordinates of **`Circle4`**. When class **`Point3`**'s constructor completes execution, it returns control to class **`Circle4`**'s constructor, which initializes the **`Circle4`**'s radius.

Software Engineering Observation 9.10

When a program creates a derived-class object, the derived-class constructor immediately calls the base-class constructor, the base-class constructor's body executes, then the derived-class constructor's body executes.

When the garbage collector removes a derived-class object from memory, the garbage collector calls that object's destructor. This begins a chain of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes execute in the reverse order of the order in which the constructors executed. Executing the destructors should free all the resources the object acquired before the garbage collector reclaims the memory for that object. When the garbage collector calls a derived-class object's destructor, the destructor performs its task, then invokes the destructor of the base class. This process repeats until class **`Object`**'s destructor is called.

C# actually implements destructors using class **`Object`**'s ***`Finalize`*** method (one of the eight methods that every C# class inherits). When compiling a class definition that contains a destructor, the compiler translates a destructor definition into a **`Finalize`** method that performs the destructor's tasks, then invokes the base class **`Finalize`** method as the last statement in the derived-class **`Finalize`** method. As mentioned in Chapter 8, we cannot determine exactly when the destructor call will occur, because we cannot determine exactly when garbage collection occurs. However, by defining a destructor, we can specify code to execute before the garbage collector removes an object from memory.

Our next example revisits the point-circle hierarchy by defining class **`Point4`** (Fig. 9.17) and class **`Circle5`** (Fig. 9.18) that contain constructors *and* destructors, each of which prints a message when it runs.

Class **`Point4`** (Fig. 9.17) contains the features shown in Fig. 9.4. We modified the constructors (lines 13–17 and 20–26) to output a line of text when they are called and added a destructor (lines 29–32) that also outputs a line of text when it is called. Each output statement (lines 16, 25 and 31) adds reference **`this`** to the output string. This implicitly invokes the class's **`ToString`** method to obtain the **`string`** representation of **`Point4`**'s coordinates.

```
1  // Fig. 9.17: Point4.cs
2  // Point4 class represents an x-y coordinate pair.
3
4  using System;
5
6  // Point4 class definition
7  public class Point4
8  {
9     // point coordinate
10    private int x, y;
11
12    // default constructor
13    public Point4()
14    {
15       // implicit call to Object constructor occurs here
16       Console.WriteLine( "Point4 constructor: {0}", this );
17    }
```

Fig. 9.17 **`Point4`** base class contains constructors and finalizer. (Part 1 of 2.)

```

   // constructor
   public Point4( int xValue, int yValue )
   {
      // implicit call to Object constructor occurs here
      X = xValue;
      Y = yValue;
      Console.WriteLine( "Point4 constructor: {0}", this );
   }

   // destructor
   ~Point4()
   {
      Console.WriteLine( "Point4 destructor: {0}", this );
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

      set
      {
         x = value; // no need for validation
      }

   } // end property X

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value; // no need for validation
      }

   } // end property Y

   // return string representation of Point4
   public override string ToString()
   {
      return "[" + x + ", " + y + "]";
   }

} // end class Point4
```

Fig. 9.17 `Point4` base class contains constructors and finalizer. (Part 2 of 2.)

Class **`Circle5`** (Fig. 9.18) contains the features in Fig. 9.13, and we modified the two constructors (lines 12–16 and 19–24) to output a line of text when they are called. We also added a destructor (lines 27–30) that also outputs a line of text when it is called. Each output statement (lines 15, 23 and 29) adds reference **`this`** to the output string. This implicitly invokes the **`Circle5`**’s **`ToString`** method to obtain the **`string`** representation of **`Circle5`**’s coordinates and radius.

Class **`ConstructorAndFinalizer`** (Fig. 9.19) demonstrates the order in which constructors and finalizers are called for objects of classes that are part of an inheritance class hierarchy. Method **`Main`** (lines 11–28) begins by instantiating an object of class **`Circle5`**, then assigns it to reference **`circle1`** (line 16). This invokes the **`Circle5`** constructor, which invokes the **`Point4`** constructor immediately. Then, the **`Point4`** constructor invokes the **`Object`** constructor. When the **`Object`** constructor (which does not print anything) returns control to the **`Point4`** constructor, the **`Point4`** constructor initializes the *x*-*y* coordinates, then outputs a **`string`** indicating that the **`Point4`** constructor was called. The output statement also calls method **`ToString`** implicitly (using reference **`this`**) to obtain the **`string`** representation of the object being constructed. Then, control returns to the **`Circle5`** constructor, which initializes the radius and outputs the **`Circle5`**’s *x*-*y* coordinates and radius by calling method **`ToString`** implicitly.

```
 1  // Fig. 9.18: Circle5.cs
 2  // Circle5 class that inherits from class Point4.
 3
 4  using System;
 5
 6  // Circle5 class definition inherits from Point4
 7  public class Circle5 : Point4
 8  {
 9     private double radius;
10
11     // default constructor
12     public Circle5()
13     {
14        // implicit call to Point3 constructor occurs here
15        Console.WriteLine( "Circle5 constructor: {0}", this );
16     }
17
18     // constructor
19     public Circle5( int xValue, int yValue, double radiusValue )
20        : base( xValue, yValue )
21     {
22        Radius = radiusValue;
23        Console.WriteLine( "Circle5 constructor: {0}", this );
24     }
25
26     // destructor overrides version in class Point4
27     ~Circle5()
28     {
29        Console.WriteLine( "Circle5 destructor: {0}", this );
30     }
```

Fig. 9.18 **`Circle5`** class inherits from class **`Point3`** and overrides a finalizer method. (Part 1 of 2.)

```

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         if ( value >= 0 )
            radius = value;
      }

   } // end property Radius

   // calculate Circle5 diameter
   public double Diameter()
   {
      return Radius * 2;
   }

   // calculate Circle5 circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle5 area
   public virtual double Area()
   {
      return Math.PI * Math.Pow( Radius, 2 );
   }

   // return string representation of Circle5
   public override string ToString()
   {
      // use base reference to return Point3 string
      return "Center = " + base.ToString() +
         "; Radius = " + Radius;
   }

} // end class Circle5
```

Fig. 9.18 `Circle5` class inherits from class `Point3` and overrides a finalizer method. (Part 2 of 2.)

Notice that the first two lines of the output from this program contain values for the *x*-*y* coordinates and the radius of `Circle5` object `circle1`. When constructing a `Circle5` object, the `this` reference used in the body of both the `Circle5` and `Point4` constructors refers to the `Circle5` object being constructed. When a program invokes method `ToString` on an object, the version of `ToString` that executes is always the version defined in that object's class. Because reference `this` refers to the current

`Circle5` object being constructed, **`Circle5`**’s **`ToString`** method executes even when **`ToString`** is invoked from the body of class **`Point4`**’s constructor. [*Note*: This would not be the case if the **`Point4`** constructor were called to initialize a an object that was actually a new **`Point4`** object.] When the **`Point4`** constructor invokes method **`ToString`** for the **`Circle5`** being constructed, the program displays **`0`** for the **`radius`** value, because the **`Circle5`** constructor’s body has not yet initialized the **`radius`**. Remember that **`0`** is the default value of a **`double`** variable. The second line of output shows the proper **`radius`** value (**`4.5`**), because that line is output after the **`radius`** is initialized.

```
// Fig. 9.19: ConstructorAndDestructor.cs
// Display order in which base-class and derived-class constructors
// and destructors are called.

using System;

// ConstructorAndFinalizer class definition
class ConstructorAndFinalizer
{
   // main entry point for application.
   static void Main( string[] args )
   {
      Circle5 circle1, circle2;

      // instantiate objects
      circle1 = new Circle5( 72, 29, 4.5 );
      circle2 = new Circle5( 5, 5, 10 );

      Console.WriteLine();

      // mark objects for garbage collection
      circle1 = null;
      circle2 = null;

      // inform garbage collector to execute
      System.GC.Collect();

   } // end method Main

  } // end class ConstructorAndDestructor
```

```
Point4 constructor: Center = [72, 29]; Radius = 0
Circle5 constructor: Center = [72, 29]; Radius = 4.5
Point4 constructor: Center = [5, 5]; Radius = 0
Circle5 constructor: Center = [5, 5]; Radius = 10

Circle5 destructor: Center = [5, 5]; Radius = 10
Point4 destructor: Center = [5, 5]; Radius = 10
Circle5 destructor: Center = [72, 29]; Radius = 4.5
Point4 destructor: Center = [72, 29]; Radius = 4.5
```

Fig. 9.19 Order in which constructors and destructors are called.

Line 17 instantiates another object of class `Circle5`, then assigns it to reference `circle2`. Again, this begins the chain of constructor calls in which the `Circle5` constructor, the `Point4` constructor and the `Object` constructor are called. In the output, notice that the body of the `Point4` constructor executes before the body of the `Circle5` constructor. This demonstrates that objects are constructed “inside out” (i.e., the base-class constructor is called first).

Lines 22–23 set references `circle1` and `circle2` to `null`. This removes the only references to these `Circle5` objects in the program. Thus, the garbage collector can release the memory that these objects occupy. Remember that we cannot guarantee when the garbage collector will execute, nor can we guarantee that it will collect all available objects when it does execute. To demonstrate the destructor invocations for the two `Circle5` objects, line 26 invokes class `GC`’s method `Collect` to request the garbage collector to run. Notice that each `Circle5` object’s destructor outputs information before calling class `Point4`’s destructor. Objects are destroyed “outside in” (i.e., the derived-class destructor completes its tasks before invoking the base-class destructor).

9.7 Software Engineering with Inheritance

In this section, we discuss the use of inheritance to customize existing software. When we use inheritance to create a new class from an existing one, the new class inherits the member variables, properties and methods of the existing class. We can customize the new class to meet our needs by including additional member variables, properties and methods, and by overriding base-class members.

Sometimes, it is difficult for students to appreciate the scope of problems faced by designers who work on large-scale software projects in industry. People experienced with such projects say that effective software reuse improves the software-development process. Object-oriented programming facilitates software reuse, thus shortening development times.

C# encourages software reuse by providing the .NET Framework Class Library (FCL), which delivers the maximum benefits of software reuse through inheritance. As interest in C# grows, interest in the FCL class libraries also increases. There is a worldwide commitment to the continued evolution of the FCL class libraries for a wide variety of applications. The FCL will grow as the .NET world grows explosively.

Software Engineering Observation 9.11

At the design stage in an object-oriented system, the designer often determines that certain classes are closely related. The designer should “factor out” common attributes and behaviors and place these in a base class. Then, use inheritance to form derived classes, endowing them with capabilities beyond those inherited from the base class.

Software Engineering Observation 9.12

The creation of a derived class does not affect its base class’s source code. Inheritance preserves the integrity of a base class.

Software Engineering Observation 9.13

Just as designers of non-object-oriented systems should avoid proliferation of functions, designers of object-oriented systems should avoid proliferation of classes. Proliferation of classes creates management problems and can hinder software reusability, because it becomes difficult for a client to locate the most appropriate class of a huge class library. The *alternative is to create fewer classes, in which each provides more substantial functionality, but such classes might provide too much functionality.*

Performance Tip 9.2

If classes produced through inheritance are larger than they need to be (i.e., contain too much functionality), memory and processing resources might be wasted. Inherit from the class whose functionality is "closest" to what is needed.

Reading derived-class definitions can be confusing, because inherited members are not shown physically in the derived class, but nevertheless are present in the derived classes. A similar problem exists when documenting derived class members.

In this chapter, we introduced inheritance—the ability to create classes by absorbing an existing class's data members and behaviors and embellishing these with new capabilities. In Chapter 10, we build upon our discussion of inheritance by introducing *polymorphism*—an object-oriented technique that enables us to write programs that handle, in a more general manner, a wide variety of classes related by inheritance. After studying Chapter 10, you will be familiar with encapsulation, inheritance and polymorphism—the most crucial aspects of object-oriented programming.

SUMMARY

- Software reusability reduces program-development time.
- The direct base class of a derived class is the base class from which the derived class inherits [via the colon (`:`) symbol]. An indirect base class of a derived class is two or more levels up the class hierarchy from that derived class.
- With single inheritance, a class is derived from one base class. C# does not support multiple inheritance (i.e., deriving a class from more than one direct base class).
- Because a derived class can include its own class variables, properties and methods, a derived class is often larger than its base class.
- A derived class is more specific than its base class and represents a smaller group of objects.
- Every object of a derived class is also an object of that class's base class. However, base-class objects are not objects of that class's derived classes.
- Derived-class methods and properties can access **`protected`** base-class members directly.
- An "is-a" relationship represents inheritance. In an "is-a" relationship, an object of a derived class also can be treated as an object of its base class.
- A "has-a" relationship represents composition. In a "has-a" relationship, a class object has references to one or more objects of other classes as members.
- A derived class cannot access **`private`** members of its base class directly.
- A derived class can access the **`public`**, **`protected`** and **`internal`** members of its base class if the derived class is in the same assembly as the base class.
- When a base-class member is inappropriate for a derived class, that member can be overridden (redefined) in the derived class with an appropriate implementation.
- To override a base-class method definition, a derived class must specify that the derived-class method overrides the base-class method with keyword **`override`** in the method header.
- Inheritance relationships form tree-like hierarchical structures. A class exists in a hierarchical relationship with its derived classes.

- It is possible to treat base-class objects and derived-class objects similarly; the commonality shared between the object types is expressed in the member variables, properties and methods of the base class.
- A base class's **`public`** members are accessible anywhere that the program has a reference to an object of that base class or to an object of one of that base class's derived classes.
- A base class's **`private`** members are accessible only within the definition of that base class.
- A base class's **`protected`** members have an intermediate level of protection between **`public`** and **`private`** access. A base class's **`protected`** members can be accessed only in that base class or in any classes derived from that base class.
- A base class's **`internal`** members can be accessed only by objects in the same assembly.
- Unfortunately, the inclusion of **`protected`** instance variables often yields two major problems. First, the derived-class object does not have to use a property to set the value of the base-class's **`protected`** data. Second, derived class methods are more likely to be written to depend on base-class implementation.
- C# rigidly enforces restriction on accessing **`private`** data members, so that even derived classes (i.e,. which are closely related to their base class) cannot access base-class **`private`** data.
- When a derived-class method overrides a base-class method, the base-class method can be accessed from the derived class by preceding the base-class method name with the **`base`** reference, followed by the dot operator (**`.`**).
- A derived class can redefine a base-class method using the same signature; this is called *overriding* that base-class method.
- A base-class method must be declared **`virtual`** if that method is to be overridden in a derived class.
- When a method is overridden in a derived class and that method is called on a derived-class object, the derived-class version (not the base-class version) is called.
- When an object of a derived class is instantiated, the base class's constructor is called immediately (either explicitly or implicitly) to do any necessary initialization of the base-class instance variables in the derived-class object (before the derived classes instance variable are initialized).
- Declaring member variables **`private`**, while providing non-**`private`** properties to manipulate and perform validation checking on this data, enforces good software engineering.
- If an object's method/property performs the actions needed by another object, call that method/property rather than duplicating its code body. Duplicated code creates code-maintenance problems.
- Base-class constructors and destructors are not inherited by derived classes.

TERMINOLOGY

abstraction
base class
base-class constructor
base-class default constructor
base-class finalizer
base-class object
base-class reference
behavior
class library
colon (**`:`**) symbol
composition
constructor
data abstraction
default constructor
derived class
derived-class constructor
derived-class reference
direct base class
dot (**`.`**) operator
garbage collector
"has-a" relationship
hierarchy diagram
indirect base class
information hiding

inheritance
inheritance hierarchy
inherited instance variable
instance variable (of an object)
`internal` member access modifier
“is-a” relationship
member-access operator
member variable (of a class)
multiple inheritance
`base` reference
`Object` class
object of a base class
object of a derived class
object-oriented programming (OOP)
overloaded constructor
overloading
`override` keyword
overriding
overriding a base-class method
overriding a method
`private` base-class member
`protected` access
`protected` base-class member
`protected` variable
`protected` member of a base class
`protected` member of a derived class
`public` member of a derived class
reusable component
single inheritance
software reusability
software reuse
`virtual` keyword

SELF-REVIEW EXERCISES

9.1 Fill in the blanks in each of the following statements:

a) __________ is a form of software reusability in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.
b) A base class’s __________ members can be accessed only in the base-class definition or in derived-class definitions.
c) In a(n) __________ relationship, an object of a derived class also can be treated as an object of its base class.
d) In a(n) __________ relationship, a class object has one or more references to objects of other classes as members.
e) A class exists in a(n) __________ relationship with its derived classes.
f) A base class’s __________ members are accessible anywhere that the program has a reference to that base class or to one of its derived classes.
g) A base class’s `protected` access members have a level of protection between those of `public` and __________ access.
h) A base class’s __________ members can be accessed only in the same assembly.
i) When an object of a derived class is instantiated, the base class’s __________ is called implicitly or explicitly to do any necessary initialization of the base-class instance variables in the derived-class object.
j) Derived-class constructors can call base-class constructors via the __________reference.

9.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) It is possible to treat base-class objects and derived-class objects similarly.
b) Base-class constructors are not inherited by derived classes.
c) A “has-a” relationship is implemented via inheritance.
d) All methods, by default, can be overridden.
e) Method `ToString` of class `System.Object` is declared as `virtual`.
f) When a derived class redefines a base-class method using the same signature, the derived class is said to overload that base-class method.
g) A `Car` class has an “is a” relationship with its `SteeringWheel` and `Brakes`.
h) Inheritance encourages the reuse of proven high-quality software.

ANSWERS TO SELF-REVIEW EXERCISES

9.1 a) Inheritance. b) **`protected`**. c) “is a” or inheritance. d) “has a” or composition or aggregation. e) hierarchical. f) **`public`**. g) **`private`**. h) **`internal`**. i) constructor. j) **`base`**.

9.2 a) True. b) True. c) False. A “has-a” relationship is implemented via composition. An “is-a” relationship is implemented via inheritance. d) False. Overridable methods must be declared explicitly as **`virtual`**. e) True. f) False. When a derived class redefines a base-class method using the same signature, the derived class overrides that base-class method. g) False. This is an example of a “has a” relationship. Class **`Car`** has an “is a” relationship with class **`Vehicle`**. h) True.

EXERCISES

9.3 Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite classes **`Point3`**, **`Circle4`** and **`Cylinder`** to use composition, rather than inheritance. After you do this, assess the relative merits of the two approaches for both the **`Point3`**, **`Circle4`**, **`Cylinder`** problem, as well as for object-oriented programs in general. Which approach is more natural, why?

9.4 Some programmers prefer not to use **`protected`** access because it breaks the encapsulation of the base class. Discuss the relative merits of using **`protected`** access vs. insisting on using **`private`** access in base classes.

9.5 Rewrite the case study in Section 9.5 as a **`Point`**, **`Square`**, **`Cube`** program. Do this two ways—once via inheritance and once via composition.

9.6 Write an inheritance hierarchy for class **`Quadrilateral`**, **`Trapezoid`**, **`Parallelogram`**, **`Rectangle`** and **`Square`**. Use **`Quadrilateral`** as the base class of the hierarchy. Make the hierarchy as deep (i.e., as many levels) as possible. The **`private`** data of **`Quadrilateral`** should be the *x*-*y* coordinate pairs for the four endpoints of the **`Quadrilateral`**. Write a program that instantiates objects of each of the classes in your hierarchy and polymorphically outputs each object’s dimensions and area.

9.7 Modify classes **`Point3`**, **`Circle4`** and **`Cylinder`** to contain destructors. Then, modify the program of Fig. 9.19 to demonstrate the order in which constructors and destructors are invoked in this hierarchy.

9.8 Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form those shapes into a shape hierarchy. Your hierarchy should have base class **`Shape`** from which class **`TwoDimensionalShape`** and class **`ThreeDimensionalShape`** are derived. Once you have developed the hierarchy, define each of the classes in the hierarchy. We will use this hierarchy in the exercises of Chapter 10 to process all shapes as objects of base-class **`Shape`**. (This is a technique called polymorphism.)

10

Object-Oriented Programming: Polymorphism

Objectives

- To understand the concept of polymorphism.
- To understand how polymorphism makes systems extensible and maintainable.
- To understand the distinction between abstract classes and concrete classes.
- To learn how to create abstract classes, interfaces and delegates.

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.
John Ronald Reuel Tolkien, *The Fellowship of the Ring*

General propositions do not decide concrete cases.
Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
Alfred North Whitehead

Outline

10.1 Introduction
10.2 Derived-Class-Object to Base-Class-Object Conversion
10.3 Type Fields and `switch` Statements
10.4 Polymorphism Examples
10.5 Abstract Classes and Methods
10.6 Case Study: Inheriting Interface and Implementation
10.7 `sealed` Classes and Methods
10.8 Case Study: Payroll System Using Polymorphism
10.9 Case Study: Creating and Using Interfaces
10.10 Delegates
10.11 Operator Overloading

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

10.1 Introduction

The previous chapter's object-oriented programming (OOP) discussion focussed on one of OOP's key component technologies, inheritance. In this chapter, we continue our study of OOP *polymorphism*. Both inheritance and polymorphism are crucial technologies in the development of complex software. Polymorphism enables us to write programs that handle a wide variety of related classes in a generic manner and facilitates adding new classes and capabilities to a system.

With polymorphism, it is possible to design and implement systems that are easily extensible. Programs can process objects of all classes in a class hierarchy generically as objects of a common base class. Furthermore, new classes can be added with little or no modification to the generic portions of the program, as long as those classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those program components that require direct knowledge of the new classes that the programmer adds to the hierarchy. In this chapter, we demonstrate two substantial class hierarchies and manipulate objects from those hierarchies polymorphically.

10.2 Derived-Class-Object to Base-Class-Object Conversion

Section 9.4 created a point-circle class hierarchy, in which class `Circle` inherited from class `Point`. The programs that manipulated objects of these classes always used `Point` references to refer to `Point` objects and `Circle` references to refer to `Circle` objects. In this section, we discuss the relationships between classes in a hierarchy that enable programs to assign derived-class objects to base-class references—a fundamental part of programs that process objects polymorphically. This section also discusses explicit casting between types in a class hierarchy.

An object of a derived class can be treated as an object of its base class. This enables various interesting manipulations. For example, a program can create an array of base-class references that refer to objects of many derived-class types. This is allowed despite the fact that the derived-class objects are of different data types. However, the reverse is not true—a base-class object is not an object of any of its derived classes. For example, a **`Point`** is not a **`Circle`** in the hierarchy defined in Chapter 9. If a base-class reference refers to a derived-class object, it is possible to convert the base-class reference to the object's actual data type and manipulate the object as that type.

The example in Fig. 10.1–Fig. 10.3 demonstrates assigning derived-class objects to base-class references and casting base-class references to derived-class references. Class **`Point`** (Fig. 10.1), which we discussed in Chapter 9, represents an *x*-*y* coordinate pair. Class **`Circle`** (Fig. 10.2), which we also discussed in Chapter 9, represents a circle and inherits from class **`Point`**. Each **`Circle`** object "is a" **`Point`** and also has a radius (represented via property **`Radius`**). We declare method **`Area`** as **`virtual`**, so that a derived class (such as class **`Cylinder`**) can override method **`Area`** to calculate the derived-class object's area. Class **`PointCircleTest`** (Fig. 10.3) demonstrates the assignment and cast operations.

```
// Fig. 10.1: Point.cs
// Point class represents an x-y coordinate pair.

using System;

// Point class definition implicitly inherits from Object
public class Point
{
   // point coordinate
   private int x, y;

   // default constructor
   public Point()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point( int xValue, int yValue )
   {
      // implicit call to Object constructor occurs here
      X = xValue;
      Y = yValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

```

Fig. 10.1 **`Point`** class represents an *x*-*y* coordinate pair. (Part 1 of 2.)

```
         set
         {
            x = value;  // no need for validation
         }

      } // end property X

      // property Y
      public int Y
      {
         get
         {
            return y;
         }

         set
         {
            y = value;  // no need for validation
         }

      } // end property Y

      // return string representation of Point
      public override string ToString()
      {
         return "[" + X + ", " + Y + "]";
      }

   } // end class Point
```

Fig. 10.1 **`Point`** class represents an *x-y* coordinate pair. (Part 2 of 2.)

```
   // Fig. 10.2: Circle.cs
   // Circle class that inherits from class Point.

   using System;

   // Circle class definition inherits from Point
   public class Circle : Point
   {
      private double radius; // circle's radius

      // default constructor
      public Circle()
      {
         // implicit call to Point constructor occurs here
      }

      // constructor
      public Circle( int xValue, int yValue, double radiusValue )
         : base( xValue, yValue )
      {
```

Fig. 10.2 **`Circle`** class that inherits from class **`Point`**. (Part 1 of 2.)

```
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         if ( value >= 0 ) // validate radius
            radius = value;
      }

   } // end property Radius

   // calculate Circle diameter
   public double Diameter()
   {
      return Radius * 2;
   }

   // calculate Circle circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle area
   public virtual double Area()
   {
      return Math.PI * Math.Pow( Radius, 2 );
   }

   // return string representation of Circle
   public override string ToString()
   {
      return "Center = " + base.ToString() +
         "; Radius = " + Radius;
   }

} // end class Circle
```

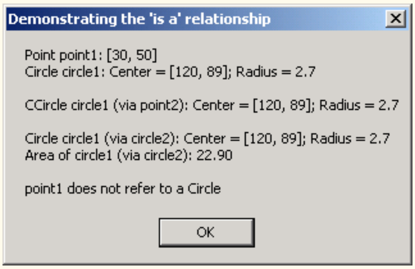
Fig. 10.2 `Circle` class that inherits from class `Point`. (Part 2 of 2.)

Class **`PointCircleTest`** (Fig. 10.3) demonstrates assigning derived-class references to base-class references and casting base-class references to derived-class references. Lines 13–14 declare a **`Point`** reference (**`point1`**) and a **`Circle`** reference (**`circle1`**). Lines 16–17 append **`String`** representations of each object to **`String output`** to show the values used to initialize these objects. Because **`point1`** is reference to a **`Point`** object, method **ToString** of **point1** prints the object as a **Point**. Similarly, because **circle1** is reference to a **Circle** object, method **ToString** of **circle1** prints the object as a **Circle**.

```
// Fig. 10.3: PointCircleTest.cs
// Demonstrating inheritance and polymorphism.

using System;
using System.Windows.Forms;

// PointCircleTest class definition
class PointCircleTest
{
   // main entry point for application.
   static void Main( string[] args )
   {
      Point point1 = new Point( 30, 50 );
      Circle circle1 = new Circle( 120, 89, 2.7 );

      string output = "Point point1: " + point1.ToString() +
         "\nCircle circle1: " + circle1.ToString();

      // use 'is a' relationship to assign
      // Circle circle1 to Point reference
      Point point2 = circle1;

      output += "\n\nCCircle circle1 (via point2): " +
         point2.ToString();

      // downcast (cast base-class reference to derived-class
      // data type) point2 to Circle circle2
      Circle circle2 = ( Circle ) point2;

      output += "\n\nCircle circle1 (via circle2): " +
         circle2.ToString();

      output += "\nArea of circle1 (via circle2): " +
         circle2.Area().ToString( "F" );

      // attempt to assign point1 object to Circle reference
      if ( point1 is Circle )
      {
         circle2 = ( Circle ) point1;
         output += "\n\ncast successful";
      }
      else
      {
         output += "\n\npoint1 does not refer to a Circle";
      }

      MessageBox.Show( output,
         "Demonstrating the 'is a' relationship" );
```

Fig. 10.3 Assigning derived-class references to base-class references. (Part 1 of 2.)

```
49
50      } // end method Main
51
52   } // end class PointCircleTest
```



```
Point point1: [30, 50]
Circle circle1: Center = [120, 89]; Radius = 2.7

CCircle circle1 (via point2): Center = [120, 89]; Radius = 2.7

Circle circle1 (via circle2): Center = [120, 89]; Radius = 2.7
Area of circle1 (via circle2): 22.90

point1 does not refer to a Circle
```

Fig. 10.3 Assigning derived-class references to base-class references. (Part 2 of 2.)

Line 21 assigns **`circle1`** (a reference to a derived-class object) to **`point2`** (a base-class reference). In C#, it is acceptable to assign a derived-class object to a base-class reference, because of the inheritance “is a” relationship. Class **`Circle`** inherits from class **`Point`**, because a **`Circle`** *is a* **`Point`** (in a structural sense, at least). However, assigning a base-class reference to a derived-class reference is potentially dangerous, as we will discuss.

Lines 23–24 invoke **`point2.ToString`** and append the result to **`output`**. When C# encounters a **`virtual`** method invocation (such as method **`ToString`**), C# determines which version of the method to call from the type of the object on which the method is called, not the type of the reference that refers to the object. In this case, **`point2`** refers to a **`Circle`** object, so C# calls **`Circle`** method **`ToString`**, rather than **`Point`** method **`ToString`** (as one might expect from the **`point2`** reference, which was declared as type **`Point`**). The decision about which method to call is an example of *polymorphism*, a concept that we discuss in detail throughout this chapter. Note that if **`point2`** referenced a **`Point`** object rather than a **`Circle`** object, C# would invoke **`Point`**’s **`ToString`** method.

Previous chapters used methods such as **`Int32.Parse`** and **`Double.Parse`** to convert between various built-in C# types. Now, we convert between object references of programmer-defined types. We use explicit casts to perform these conversions. If the cast is valid, our program can treat a base-class reference as a derived-class reference. If the cast is invalid, C# throws an ***`InvalidCastException`***, which indicates that the cast operation is not allowed. Exceptions are discussed in detail in Chapter 11, Exception Handling.

Common Programming Error 10.1

Assigning a base-class object (or a base-class reference) to a derived-class reference (without an explicit cast) is a syntax error.

Software Engineering Observation 10.1

If a derived-class object has been assigned to a reference of one of its direct or indirect base classes, it is acceptable to cast that base-class reference back to a reference of the derived-class type. In fact, this must be done to send that object messages that do not appear in the base class. [Note: We sometimes use the term “messages” to represent the invocation of methods and the use of object properties.]

Line 28 casts **`point2`**, which currently refers to a **`Circle`** object (**`circle1`**), to a **`Circle`** and assigns the result to **`circle2`**. As we discuss momentarily, this cast would be dangerous if **`point2`** were referencing a **`Point`**. Lines 30–31 invoke method **`ToString`** of the **`Circle`** object to which **`circle2`** now refers (note that the fourth line of the output demonstrates that **`Circle`**'s **`ToString`** method is called). Lines 33–34 calculate and output **`circle2`**'s **`Area`**.

Line 39 explicitly casts reference **`point1`** to a **`Circle`**. This is a dangerous operation, because **`point`** refers to a **`Point`** object, and a **`Point`** is not a **`Circle`**. Objects can be cast only to their own type or to their base-class types. If this statement were to execute, C# would determine that **`point1`** references a **`Point`** object, recognize the cast to **`Circle`** as dangerous and indicate an improper cast with an **`InvalidCastException`** message. However, we prevent this statement from executing by including an **`if`**/**`else`** structure (lines 37–45). The condition at line 37 uses keyword **`is`** to determine whether the object to which **`point1`** refers "is a" **`Circle`**. Keyword **`is`** discovers the type of the object to which the left operand refers and compares this type to the right operand (in this case, **`Circle`**). In our example, **`point1`** does not refer to a **`Circle`**, so the condition fails, and line 44 appends to **`output`** a string that indicates the result. Note that the **`is`** comparison will be **`true`** if the left operand is a reference to an instance of the right operand or a derived class.

Common Programming Error 10.2

Attempting to cast a base-class reference to a derived-class type causes an ***`InvalidCastException`*** *if the reference refers to a base-class object rather than an appropriate derived-class object.*

Software Engineering Observation 10.2

The ***`is`*** *keyword enables a program to determine whether a cast operation would be successful by ensuring that the reference type and target type are compatible.*

If we remove the **`if`** test and execute the program, C# displays a **`MessageBox`** that contains the message:

```
An unhandled exception of type 'System.InvalidCastException'
occurred in
```

followed by the name and path of the executing program. We discuss how to deal with such situations in Chapter 11.

Despite the fact that a derived-class object also "is a" base-class object, the derived-class and base-class objects are different. As we have discussed previously, derived-class objects can be treated as if they were base-class objects. This is a logical relationship, because the derived class contains members that correspond to all members in the base class, but the derived class can have additional members. For this reason, assigning a base-class object to a derived-class reference is not allowed without an explicit cast. Such an assignment would leave the additional derived-class members undefined.

There are four ways to mix base-class references and derived-class references with base-class objects and derived-class objects:

1. Referring to a base-class object with a base-class reference is straightforward.
2. Referring to a derived-class object with a derived-class reference is straightforward.

3. Referring to a derived-class object with a base-class reference is safe, because the derived-class object *is an* object of its base class. However, this reference can refer only to base-class members. If this code refers to derived-class-only members through the base-class reference, the compiler reports an error.
4. Referring to a base-class object with a derived-class reference generates a compiler error. To avoid this error, the derived-class reference first must be cast to a base-class reference explicitly. In this cast, the derived-class reference must reference a derived-class object, or C# generates an `InvalidCastException`.

Common Programming Error 10.3

After assigning a derived-class object to a base-class reference, attempting to reference derived-class-only members with the base-class reference is a compilation error.

Common Programming Error 10.4

Treating a base-class object as a derived-class object can cause errors.

Though it is convenient to treat derived-class objects as base-class objects by manipulating derived-class objects with base-class references, doing so can cause significant problems. For example, a payroll system, must be able to traverse an array of employees and calculate the weekly pay for each person. Intuition suggests that using base-class references would enable the program to call only the base-class payroll calculation routine (if there is such a routine in the base class). Using only base-class references, we can invoke the proper payroll calculation routine for each object, whether the object is a base-class object or a derived-class object. We learn how to create classes that exhibit this behavior as we introduce polymorphism throughout this chapter.

10.3 Type Fields and `switch` Statements

One way to determine the type of an object that is incorporated in a larger program is to use a `switch` structure. This allows us to distinguish among object types, then invoke an appropriate action for a particular object. For example, in a hierarchy of shapes in which each shape object has a `ShapeType` property, a `switch` structure could employ the object's `ShapeType` to determine which `Print` method to call.

However, using `switch` logic exposes programs to a variety of potential problems. For example, the programmer might forget to include a type test when one is warranted, or the programmer might forget to test all possible cases in a `switch` structure. When modifying a `switch`-based system by adding new types, the programmer might forget to insert the new cases in all relevant `switch` statements. Every addition or deletion of a class requires the modification of every `switch` statement in the system; tracking these statements down can be time consuming and error prone.

Software Engineering Observation 10.3

Polymorphic programming can eliminate the need for unnecessary `switch` logic. By using C#'s polymorphism mechanism to perform the equivalent logic, programmers can avoid the kinds of errors typically associated with `switch` logic.

Testing and Debugging Tip 10.1

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and more simple, sequential code. This simplification facilitates testing, debugging and program maintenance.

10.4 Polymorphism Examples

In this section, we discuss several examples of polymorphism. If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object. Any operation (such as calculating the perimeter or the area) that can be performed on an object of class `Quadrilateral` also can be performed on an object of class `Rectangle`. Such operations also can be performed on other kinds of `Quadrilateral`s, such as `Square`s, `Parallelogram`s and `Trapezoid`s. When a program invokes a derived-class method through a base-class (i.e., `Quadrilateral`) reference, C# polymorphically chooses the correct overriding method in the derived class from which the object was instantiated. We investigate this behavior in later examples.

Suppose that we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venutian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Also, imagine that each of these classes inherits from the common base class called `SpaceObject`, which contains method `DrawYourself`. Each derived class implements this method. A screen-manager program would maintain a container (such as a `SpaceObject` array) of references to objects of the various classes. To refresh the screen, the screen manager would periodically send each object the same message—namely, `DrawYourself`. However, each object responds in a unique way. For example, a `Martian` object would draw itself in red with the appropriate number of antennae. A `SpaceShip` object would draw itself as a bright, silver flying saucer. A `LaserBeam` object would draw itself as a bright red beam across the screen. Thus the same message sent to a variety of objects would have “many forms” of results—hence the term *polymorphism*.

A polymorphic screen manager facilitates adding new classes to a system with minimal modifications to the system’s code. Suppose we want to add class `Mercurian`s to our video game. To do so, we must build a class `Mercurian` that inherits from `SpaceObject`, but provides its own definition of method `DrawYourself`. Then, when objects of class `Mercurian` appear in the container, the programmer does not need to modify the code for the screen manager. The screen manager invokes method `DrawYourself` on every object in the container, regardless of the object’s type, so the new `Mercurian` objects simply “plug right in.” Thus, without modifying the system (other than to build and include the classes themselves), programmers can use polymorphism to include additional types of classes that were not envisioned when the system was created.

With polymorphism, one method can cause different actions to occur, depending on the type of the object on which the method is invoked. This gives the programmer tremendous expressive capability. In the next several sections, we provide examples that demonstrate polymorphism.

Software Engineering Observation 10.4

With polymorphism, the programmer can deal in generalities and let the execution-time environment concern itself with the specifics. The programmer can command a wide variety of objects to behave in manners appropriate to those objects, even if the programmer does not know the objects’ types.

Software Engineering Observation 10.5

Polymorphism promotes extensibility. Software used to invoke polymorphic behavior is written to be independent of the types of the objects to which messages (i.e., method calls) are sent. Thus, programmers can include into a system additional types of objects that respond to existing messages and can do this without modifying the base system.

10.5 Abstract Classes and Methods

When we think of a class as a type, we assume that programs will create objects of that type. However, there are cases in which it is useful to define classes for which the programmer never intends to instantiate any objects. Such classes are called *abstract classes*. Because such classes normally are used as base classes in inheritance hierarchies, we refer to such classes as *abstract base classes*. These classes cannot be used to instantiate objects, since abstract classes are incomplete. Derived classes must define the “missing pieces.” Abstract classes normally contain one or more *abstract methods* or *abstract properties*, which are methods and properties that do not provide implementations. Derived classes must override inherited abstract methods and properties to enable objects of those derived classes to be instantiated. We discuss abstract classes extensively in Section 10.6 and Section 10.8.

The purpose of an abstract class is to provide an appropriate base class from which other classes may inherit. Classes from which objects can be instantiated are called *concrete classes*. Such classes provide implementations of every method and property they define. We could have an abstract base class `TwoDimensionalObject` and derive such concrete classes as `Square`, `Circle` and `Triangle`. We could also have an abstract base class `ThreeDimensionalObject` and derive such concrete classes as `Cube`, `Sphere` and `Cylinder`. Abstract base classes are too generic to define real objects; we need to be more specific before we can think of instantiating objects. For example, if someone tells you to “draw the shape,” what shape would you draw? Concrete classes provide the specifics that make it reasonable to instantiate objects.

A class is made abstract by declaring it with keyword ***`abstract`***. An inheritance hierarchy does not need to contain any abstract classes, but, as we will see, many good object-oriented systems have class hierarchies headed by abstract base classes. In some cases, abstract classes constitute the top few levels of the hierarchy. A good example of this is the shape hierarchy in Fig. 9.3. The hierarchy begins with abstract base-class `Shape`. On the next level of the hierarchy, we have two more abstract base classes, namely `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy would define concrete classes for two-dimensional shapes, such as `Circle` and `Square`, and for three-dimensional shapes, such as `Sphere` and `Cube`.

Software Engineering Observation 10.6

An abstract class defines a common set of `public` *methods and properties for the various members of a class hierarchy. An abstract class typically contains one or more abstract methods and properties that derived classes will override. All classes in the hierarchy can use this common set of* `public` *methods and properties.*

Abstract classes must specify signatures for their abstract methods and properties. C# provides keyword ***`abstract`*** to declare a method or property as abstract. Methods and properties that are `abstract` do not provide implementations—attempting to do so is a syntax error. Every concrete derived class must override all base-class `abstract` methods and properties (using keyword `override`) and provide concrete implementations of those methods or properties. Any class with an `abstract` method in it must be declared `abstract`. The difference between an `abstract` method and a `virtual` method is that a `virtual` method has an implementation and provides the derived class with the option of overriding the method; by contrast, an `abstract` method does not provide an implementation and forces the derived class to override the method (for that derived class to be concrete).

Common Programming Error 10.5

Defining an `abstract` method in a class that has not been declared as `abstract` results is a syntax error.

Common Programming Error 10.6

Attempting to instantiate an object of an `abstract` class results in a compilation error.

Common Programming Error 10.7

Failure to override an `abstract` method in a derived class is a syntax error, unless the derived class also is an `abstract` class.

Software Engineering Observation 10.7

An `abstract` class can have instance data and non-`abstract` methods (including constructors), which are subject to the normal rules of inheritance by derived classes.

Although we cannot instantiate objects of abstract base classes, we *can* use abstract base classes to declare references; these references can refer to instances of any concrete classes derived from the abstract class. Programs can use such references to manipulate instances of the derived classes polymorphically.

Let us consider another application of polymorphism. A screen manager needs to display a variety of objects, including new types of objects that the programmer will add to the system after writing the screen manager. The system might need to display various shapes, such as `Circle`, `Triangle` or `Rectangle`, which are derived from abstract class `Shape`. The screen manager uses base-class references of type `Shape` to manage the objects that are displayed. To draw any object (regardless of the level at which that object's class appears in the inheritance hierarchy), the screen manager uses a base-class reference to the object to invoke the object's `Draw` method. Method `Draw` is an `abstract` method in base class `Shape`; therefore, each derived class must implement method `Draw`. Each `Shape` object in the inheritance hierarchy knows how to draw itself. The screen manager does not have to worry about the type of each object or whether the screen manager has ever encountered objects of that type.

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to *read* or *write* data from and to devices may have a certain uniformity. The write message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The capabilities (i.e., the **`public`** services) offered by the device drivers are provided as abstract methods in the abstract base class. The implementations of these abstract methods are provided in the derived classes that correspond to the specific types of device drivers.

It is common in object-oriented programming to define an *iterator class* that can traverse all the objects in a container (such as an array). For example, a program can print a list of objects in a linked list by creating an iterator object, then using the iterator to obtain the next element of the list each time the iterator is called. Iterators often are used in polymorphic programming to traverse an array or a linked list of objects from various levels of a hierarchy. The references in such a list are all base-class references. (See Chapter 23, Data Structures, to learn more about linked lists.) A list of objects of base class **`TwoDimensionalShape`** could contain objects from classes **`Square`**, **`Circle`**, **`Triangle`** and so on. Using polymorphism to send a **`Draw`** message to each object in the list would draw each object correctly on the screen.

10.6 Case Study: Inheriting Interface and Implementation

Our next example (Fig. 10.4–Fig. 10.8) reexamines the **`Point`**, **`Circle`**, **`Cylinder`** hierarchy that we explored in Chapter 9. In this example, the hierarchy begins with abstract base class **`Shape`** (Fig. 10.4). This hierarchy mechanically demonstrates the power of polymorphism. In the exercises, we explore a more substantial shape hierarchy.

```
// Fig. 10.4: Shape.cs
// Demonstrate a shape hierarchy using an abstract base class.
using System;

public abstract class Shape
{
   // return Shape's area
   public virtual double Area()
   {
      return 0;
   }

   // return Shape's volume
   public virtual double Volume()
   {
      return 0;
   }

   // return Shape's name
   public abstract string Name
   {
      get;
   }
}
```

Fig. 10.4 Abstract **`Shape`** base class.

Class **`Shape`** defines two concrete methods and one **`abstract`** property. All shapes have an area and a volume, so we include virtual methods **`Area`** (lines 8–11) and **`Volume`** (lines 14–17), which return the shape's area and volume, respectively. The volume of two-dimensional shapes is always zero, whereas three-dimensional shapes have a positive, nonzero volume. In class **`Shape`**, methods **`Area`** and **`Volume`** return zero, by default. Programmers can override these methods in derived classes when those classes should have different area calculations [e.g., classes **`Circle2`** (Fig. 10.6) and **`Cylinder2`** (Fig. 10.7)] and/or different volume calculations (e.g., **`Cylinder2`**). Read-only property **`Name`** (lines 20–23) is declared **`abstract`**, so derived classes must implement this property to become concrete classes. Note that **`abstract`** methods and properties are implicitly **`virtual`**.

Class **`Point2`** (Fig. 10.5) inherits from **`abstract`** class **`Shape`** and overrides the **`abstract`** property **`Name`**, which makes **`Point2`** a concrete class. A point's area and volume are zero, so class **`Point2`** does not override base-class methods **`Area`** and **`Volume`**. Lines 59–65 implement property **`Name`**. If we did not provide this implementation, class **`Point2`** would be an abstract class that would require keyword **`abstract`** in the first line of the class definition.

```
1  // Fig. 10.5: Point2.cs
2  // Point2 inherits from abstract class Shape and represents
3  // an x-y coordinate pair.
4  using System;
5
6  // Point2 inherits from abstract class Shape
7  public class Point2 : Shape
8  {
9     private int x, y; // Point2 coordinates
10
11    // default constructor
12    public Point2()
13    {
14       // implicit call to Object constructor occurs here
15    }
16
17    // constructor
18    public Point2( int xValue, int yValue )
19    {
20       X = xValue;
21       Y = yValue;
22    }
23
24    // property X
25    public int X
26    {
27       get
28       {
29          return x;
30       }
31
```

Fig. 10.5 **`Point2`** class inherits from **`abstract`** class **`Shape`**. (Part 1 of 2.)

```
      set
      {
         x = value; // no validation needed
      }
   }

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value; // no validation needed
      }
   }

   // return string representation of Point2 object
   public override string ToString()
   {
      return "[" + X + ", " + Y + "]";
   }

   // implement abstract property Name of class Shape
   public override string Name
   {
      get
      {
         return "Point2";
      }
   }

} // end class Point2
```

Fig. 10.5 `Point2` class inherits from `abstract` class `Shape`. (Part 2 of 2.)

Figure 10.6 defines class **`Circle2`**, which inherits from class **`Point2`**. Class **`Circle2`** contains property **`Radius`** (lines 24–37) for accessing the circle's radius. Note that we do not declare property **`Radius`** as **`virtual`**, so classes derived from this class cannot override this property. A circle has zero volume, so we do not override base-class method **`Volume`**. Rather, **`Circle2`** inherits this method from class **`Point2`**, which inherited the method from **`Shape`**. However, a circle does have an area, so **`Circle2`** overrides **`Shape`** method **`Area`** (lines 52–55). Property **`Name`** (lines 65–71) of class **`Circle2`** overrides property **`Name`** of class **`Point2`**. If this class did not override property **`Name`**, the class would inherit the **`Point2`** version of property **`Name`**. In that case, **`Circle2`**'s **`Name`** property would erroneously return "**`Point2`**."

```
// Fig. 10.6: Circle2.cs
// Circle2 inherits from class Point2 and overrides key members.
using System;

// Circle2 inherits from class Point2
public class Circle2 : Point2
{
   private double radius; // Circle2 radius

   // default constructor
   public Circle2()
   {
      // implicit call to Point2 constructor occurs here
   }

   // constructor
   public Circle2( int xValue, int yValue, double radiusValue )
      : base( xValue, yValue )
   {
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         // ensure non-negative radius value
         if ( value >= 0 )
            radius = value;
      }
   }

   // calculate Circle2 diameter
   public double Diameter()
   {
      return Radius * 2;
   }

   // calculate Circle2 circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle2 area
   public override double Area()
   {
```

Fig. 10.6 `Circle2` class that inherits from class `Point2`. (Part 1 of 2.)

```
         return Math.PI * Math.Pow( Radius, 2 );
      }

      // return string representation of Circle2 object
      public override string ToString()
      {
         return "Center = " + base.ToString() +
            "; Radius = " + Radius;
      }

      // override property Name from class Point2
      public override string Name
      {
         get
         {
            return "Circle2";
         }
      }

   } // end class Circle2
```

Fig. 10.6 `Circle2` class that inherits from class `Point2`. (Part 2 of 2.)

Figure 10.7 defines class **`Cylinder2`**, which inherits from class **`Circle2`**. Class **`Cylinder2`** contains property **`Height`** (lines 24–37) for accessing the cylinder's height. Note that we do not declare property **`Height`** as **`virtual`**, so classes derived from class **`Cylinder2`** cannot override this property. A cylinder has different area and volume calculations from those of a circle, so this class overrides method **`Area`** (lines 40–43) to calculate the cylinder's surface area (i.e., $2\pi r^2 + 2\pi rh$) and overrides method **`Volume`** (lines 46–49). Property **`Name`** (lines 58–64) overrides property **`Name`** of class **`Circle2`**. If this class did not override property **`Name`**, the class would inherit property **`Name`** of class **`Circle2`**, and this property would erroneously return "**`Circle2`**."

```
1  // Fig. 10.7: Cylinder2.cs
2  // Cylinder2 inherits from class Circle2 and overrides key members.
3  using System;
4
5  // Cylinder2 inherits from class Circle2
6  public class Cylinder2 : Circle2
7  {
8     private double height; // Cylinder2 height
9
10    // default constructor
11    public Cylinder2()
12    {
13       // implicit call to Circle2 constructor occurs here
14    }
15
```

Fig. 10.7 `Cylinder2` class inherits from class `Circle2`. (Part 1 of 2.)

```
   // constructor
   public Cylinder2( int xValue, int yValue, double radiusValue,
      double heightValue ) : base( xValue, yValue, radiusValue )
   {
      Height = heightValue;
   }

   // property Height
   public double Height
   {
      get
      {
         return height;
      }

      set
      {
         // ensure non-negative height value
         if ( value >= 0 )
            height = value;
      }
   }

   // calculate Cylinder2 area
   public override double Area()
   {
      return 2 * base.Area() + base.Circumference() * Height;
   }

   // calculate Cylinder2 volume
   public override double Volume()
   {
      return base.Area() * Height;
   }

   // return string representation of Circle2 object
   public override string ToString()
   {
      return base.ToString() + "; Height = " + Height;
   }

   // override property Name from class Circle2
   public override string Name
   {
      get
      {
         return "Cylinder2";
      }
   }

} // end class Cylinder2
```

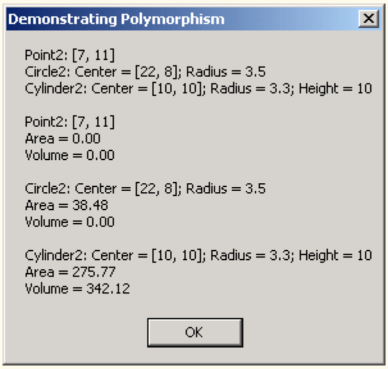
Fig. 10.7 **`Cylinder2`** class inherits from class **`Circle2`**. (Part 2 of 2.)

Class **`Test2`** (Fig. 10.8), creates an object of each of the three concrete classes and manipulates those objects polymorphically using an array of **`Shape`** references. Lines 11–13 instantiate **`Point2`** object **`point`**, **`Circle2`** object **`circle`**, and **`Cylinder2`** object **`cylinder`**, respectively. Next, line 16 allocates array **`arrayOfShapes`**, which contains three **`Shape`** references. Line 19 assigns reference **`point`** to the array element **`arrayOfShapes[ 0 ]`**, line 22 assigns reference **`circle`** to the array element **`arrayOfShapes[ 1 ]`** and line 25 assigns reference **`cylinder`** to the array element **`arrayOfShapes[ 2 ]`**. These assignments are possible because a **`Point2`** is a **`Shape`**, a **`Circle2`** is a **`Shape`** and a **`Cylinder2`** is a **`Shape`**. Therefore, we can assign instances of derived classes **`Point2`**, **`Circle2`** and **`Cylinder2`** to base-class **`Shape`** references.

```
 1  // Fig. 10.8: AbstractShapesTest.cs
 2  // Demonstrates polymorphism in Point-Circle-Cylinder hierarchy.
 3  using System;
 4  using System.Windows.Forms;
 5
 6  public class AbstractShapesTest
 7  {
 8     public static void Main( string[] args )
 9     {
10        // instantiate Point2, Circle2 and Cylinder2 objects
11        Point2 point = new Point2( 7, 11 );
12        Circle2 circle = new Circle2( 22, 8, 3.5 );
13        Cylinder2 cylinder = new Cylinder2( 10, 10, 3.3, 10 );
14
15        // create empty array of Shape base-class references
16        Shape[] arrayOfShapes = new Shape[ 3 ];
17
18        // arrayOfShapes[ 0 ] refers to Point2 object
19        arrayOfShapes[ 0 ] = point;
20
21        // arrayOfShapes[ 1 ] refers to Circle2 object
22        arrayOfShapes[ 1 ] = circle;
23
24        // arrayOfShapes[ 1 ] refers to Cylinder2 object
25        arrayOfShapes[ 2 ] = cylinder;
26
27        string output = point.Name + ": " + point + "\n" +
28           circle.Name + ": " + circle + "\n" +
29           cylinder.Name + ": " + cylinder;
30
31        // display Name, Area and Volume for each object
32        // in arrayOfShapes polymorphically
33        foreach( Shape shape in arrayOfShapes )
34        {
35           output += "\n\n" + shape.Name + ": " + shape +
36              "\nArea = " + shape.Area().ToString( "F" ) +
37              "\nVolume = " + shape.Volume().ToString( "F" );
38        }
39
```

Fig. 10.8 **`AbstractShapesTest`** demonstrates polymorphism in Point-Circle-Cylinder hierarchy. (Part 1 of 2.)

```
40          MessageBox.Show( output, "Demonstrating Polymorphism" );
41       }
42    }
```



```
Point2: [7, 11]
Circle2: Center = [22, 8]; Radius = 3.5
Cylinder2: Center = [10, 10]; Radius = 3.3; Height = 10

Point2: [7, 11]
Area = 0.00
Volume = 0.00

Circle2: Center = [22, 8]; Radius = 3.5
Area = 38.48
Volume = 0.00

Cylinder2: Center = [10, 10]; Radius = 3.3; Height = 10
Area = 275.77
Volume = 342.12
```

Fig. 10.8 `AbstractShapesTest` demonstrates polymorphism in Point-Circle-Cylinder hierarchy. (Part 2 of 2.)

Lines 27–29 access property **`Name`** and invoke method **`ToString`** (implicitly) for objects **`point`**, **`circle`** and **`cylinder`**. Property **`Name`** returns the object's class name and method **`ToString`** returns the object's **`String`** representation (i.e., *x-y* coordinate pair, radius and height, depending on each object's type). Note that lines 27–29 use derived-class references to invoke each derived-class object's methods and properties.

By contrast, the **`foreach`** structure (lines 33–38) uses base-class **`Shape`** references to invoke each derived-class object's methods and properties. The **`foreach`** structure calls property **`Name`** and methods **`ToString`**, **`Area`** and **`Volume`** for each **`Shape`** reference in **`arrayOfShapes`**. The property and methods are invoked on each object in **`arrayOfShapes`**. When the compiler looks at each method/property call, the compiler determines whether each **`Shape`** reference (in **`arrayOfShapes`**) can make these calls. This is the case for property **`Name`** and methods **`Area`** and **`Volume`**, because they are defined in class **`Shape`**. However, class **`Shape`** does not define method **`ToString`**. For this method, the compiler proceeds to **`Shape`**'s base class (class **`Object`**) and determines that **`Shape`** inherited a no-argument **`ToString`** method from class **`Object`**.

The screen capture of Fig. 10.8 illustrates that the "appropriate" property **`Name`** and methods **`ToString`**, **`Area`** and **`Volume`** were invoked for each type of object in **`arrayOfShapes`**. By "appropriate," we mean that C# maps each property and method call to the proper object. For example, in the **`foreach`** structure's first iteration, reference **`arrayOfShapes[ 0 ]`** (which is of type **`Shape`**) refers to the same object as **`point`** (which is of type **`Point2`**). Class **`Point2`** overrides property **`Name`**, and method **`ToString`** and inherits method **`Area`** and **`Volume`** from class **`Shape`**. At runtime, **`arrayOfShapes[ 0 ]`** accesses property **`Name`** and invokes methods **`ToString`**, **`Area`** and **`Volume`** of the **`Point`** object. C# determines the correct object type, then uses that type to determine the appropriate methods to invoke. Through polymorphism, the call to property **`Name`** returns the string **`"Point2:"`**; the call to method **`ToString`** returns the **`String`** representation of **`point`**'s *x-y* coordinate pair; and methods **`Area`** and **`Volume`** each return **`0`** (as shown in the second group of outputs in Fig. 10.8).

Polymorphism occurs in the next two iterations of the `foreach` structure as well. Reference `arrayOfShapes[ 1 ]` refers to the same object as `circle` (which is of type `Circle2`). Class `Circle2` provides implementations for property `Name`, method `ToString` and method `Area`, and inherits method `Volume` from class `Point2` (which, in turn, inherited method `Volume` from class `Shape`). C# associates property `Name` and methods `ToString`, `Area` and `Volume` of the `Circle2` object to reference `arrayOfShapes[ 1 ]`. As a result, property `Name` returns the string `"Circle2:"`; method `ToString` returns the `String` representation of `circle`'s *x*-*y* coordinate pair and radius; method `Area` returns the area (`38.48`); and method `Volume` returns `0`.

For the final iteration of the `foreach` structure, reference `arrayOfShapes[ 2 ]` refers to the same object as `cylinder` (which is of type `Cylinder2`). Class `Cylinder2` provides its own implementations for property `Name` and for methods `ToString`, `Area` and `Volume`. C# associates property `Name` and methods `ToString`, `Area` and `Volume` of the `Cylinder2` object to reference `arrayOfShapes[ 2 ]`. Property `Name` returns the string `"Cylinder2:"`; method `ToString` returns the `String` representation of `cylinder`'s *x*-*y* coordinate pair, radius and height; method `Area` returns the cylinder's surface area (`275.769`…); and method `Volume` returns the cylinder's volume (`342.119`…).

10.7 `sealed` Classes and Methods

In Chapter 8, Object-Based Programming, we saw that variables can be declared `const` and `readonly` to indicate that they cannot be modified after they are initialized. Variables declared with `const` must be initialized when they are declared; variables declared with `readonly` can be initialized in the constructor, but cannot be changed after they are initialized.

The keyword ***`sealed`*** provides is applied to methods and classes to prevent overriding and inheritance. A method that is declared `sealed` cannot be overridden in a derived class. Methods that are declared `static` and methods that are declared `private` are implicitly `sealed`.

Performance Tip 10.1

The `sealed` keyword allows certain compiler optimizations. A `sealed` method's definition can never change, so the compiler can optimize the program by removing calls to `sealed` methods and replacing them with the expanded code of their definitions at each method call location—a technique known as inlining the code.

Software Engineering Observation 10.8

If a method is declared `sealed`, it cannot be overridden in derived classes. Method calls must not be sent polymorphically to objects of those derived classes. The method call still may be sent to derived classes, but they will respond identically, rather than polymorphically. Remember that a method cannot be overridden (using the keyword `override`) if it is not declared either `virtual` or `abstract`. Therefore, keyword `sealed` is not needed for these cases. Keyword `sealed` is used for methods that have been overridden, but that we do not want to be overridden in derived classes.

Performance Tip 10.2

The compiler can decide to inline a `sealed` method call and will do so for small, simple `sealed` methods. Inlining does not violate encapsulation or information hiding (but does improve performance, because it eliminates the overhead of making a method call).

Performance Tip 10.3

Pipelined processors can improve performance by executing portions of the next several instructions simultaneously, but not if those instructions follow a method call. Inlining (which the compiler can perform on a `sealed` *method) can improve performance in these processors as it eliminates the out-of-line transfer of control associated with a method call.*

Software Engineering Observation 10.9

A class that is declared `sealed` *cannot be a base class (i.e., a class cannot inherit from a* `sealed` *class). All methods in a* `sealed` *class are* `sealed` *implicitly.*

Using the `sealed` keyword with classes allows other runtime optimizations. For example, `virtual` method calls can be transformed into non-`virtual` method calls.

A `sealed` class is the opposite of an `abstract` class in certain ways. An `abstract` class cannot be instantiated—other classes derive from the abstract base class and implement the base class's `abstract` members. A `sealed` class, on the other hand, cannot have any derived classes. This relationship is similar with regard to methods. An `abstract` method must be overridden in a derived class. A `sealed` method cannot be overridden in a derived class.

10.8 Case Study: Payroll System Using Polymorphism

Let us use `abstract` classes, `abstract` methods and polymorphism to perform payroll calculations for various types of employees. We begin by creating abstract base class `Employee`. The derived classes of `Employee` are `Boss` (paid a fixed weekly salary, regardless of the number of hours worked), `CommissionWorker` (paid a flat base salary plus a percentage of the worker's sales), `PieceWorker` (paid a flat fee per item produced) and `HourlyWorker` (paid by the hour with "time-and-a-half" for overtime).

The application must determine the weekly earnings for all types of employees, so each class derived from `Employee` requires method `Earnings`. However, each derived class uses a different calculation to determine earnings for each specific type of employee. Therefore, we declare method `Earnings` as `abstract` in `Employee` and declare `Employee` to be an `abstract` class. Each derived class overrides this method to calculate earnings for that employee type.

To calculate any employee's earnings, the program can use a base-class reference to a derived-class object and invoke method `Earnings`. A real payroll system might reference the various `Employee` objects with individual elements in an array of `Employee` references. The program would traverse the array one element at a time, using the `Employee` references to invoke the appropriate `Earnings` method of each object.

Software Engineering Observation 10.10

The ability to declare an abstract method gives the class designer considerable control over how derived classes are defined in a class hierarchy. Any class that inherits directly from a base class containing an abstract method must override the abstract method. Otherwise, the new class also would be abstract, and attempts to instantiate objects of that class would fail.

Let us consider class `Employee` (Fig. 10.9). The `public` members include a constructor (lines 11–16) that takes as arguments the employee's first and last names; properties `FirstName` (lines 19–30) and `LastName` (lines 33–44); method `ToString` (lines 47–50), which returns the first name and last name separated by a space; and `abstract` method **`Earnings`** (line 54). The **`abstract`** keyword (line 5) indicates that class **`Employee`** is abstract; thus, it cannot be used to instantiate **`Employee`** objects. Method **`Earnings`** is declared **`abstract`**, so the class does not provide a method implementation. All classes derived directly from class **`Employee`**—except for abstract derived classes—must implement this method. Method **`Earnings`** is **`abstract`** in **`Employee`**, because we cannot calculate the earnings for a generic employee. To determine earnings, we first must know of what *kind* the employee is. By declaring this method **`abstract`**, we indicate that we will provide an implementation in each concrete derived class, but not in the base class itself.

```
 1  // Fig. 10.9: Employee.cs
 2  // Abstract base class for company employees.
 3  using System;
 4
 5  public abstract class Employee
 6  {
 7     private string firstName;
 8     private string lastName;
 9
10     // constructor
11     public Employee( string firstNameValue,
12        string lastNameValue )
13     {
14        FirstName = firstNameValue;
15        LastName = lastNameValue;
16     }
17
18     // property FirstName
19     public string FirstName
20     {
21        get
22        {
23           return firstName;
24        }
25
26        set
27        {
28           firstName = value;
29        }
30     }
31
32     // property LastName
33     public string LastName
34     {
35        get
36        {
37           return lastName;
38        }
39
```

Fig. 10.9 **`abstract`** class **`Employee`** definition. (Part 1 of 2.)

```
         set
         {
            lastName = value;
         }
      }

      // return string representation of Employee
      public override string ToString()
      {
         return FirstName + " " + LastName;
      }

      // abstract method that must be implemented for each derived
      // class of Employee to calculate specific earnings
      public abstract decimal Earnings();

   } // end class Employee
```

Fig. 10.9 `abstract` class `Employee` definition. (Part 2 of 2.)

Class **`Boss`** (Fig. 10.10) inherits from **`Employee`**. Class **`Boss`**’s constructor (lines 10–15) receives as arguments a first name, a last name and a salary. The constructor passes the first name and last name to the **`Employee`** constructor (line 12), which initializes the **`FirstName`** and **`LastName`** members of the base-class part of the derived-class object. Other **`public`** methods in class **`Boss`** include method **`Earnings`** (lines 34–37), which defines the calculation of a boss’ earnings, and method **`ToString`** (lines 40–43), which returns a string that indicates the type of employee (i.e., **`"Boss: "`**) and the boss’s name. Class **`Boss`** also includes property **`WeeklySalary`** (lines 18–31), which manipulates the value for member variable **`salary`**. Note that this property ensures only that **`salary`** cannot hold a negative value—in a real payroll system, this validation would be more extensive and carefully controlled.

```
 1   // Fig. 10.10: Boss.cs
 2   // Boss class derived from Employee.
 3   using System;
 4
 5   public class Boss : Employee
 6   {
 7      private decimal salary; // Boss's salary
 8
 9      // constructor
10      public Boss( string firstNameValue, string lastNameValue,
11         decimal salaryValue)
12         : base( firstNameValue, lastNameValue )
13      {
14         WeeklySalary = salaryValue;
15      }
16
```

Fig. 10.10 `Boss` class inherits from class `Employee`. (Part 1 of 2.)

```
   // property WeeklySalary
   public decimal WeeklySalary
   {
      get
      {
         return salary;
      }

      set
      {
         // ensure positive salary value
         if ( value > 0 )
            salary = value;
      }
   }

   // override base-class method to calculate Boss's earnings
   public override decimal Earnings()
   {
      return WeeklySalary;
   }

   // return string representation of Boss
   public override string ToString()
   {
      return "Boss: " + base.ToString();
   }
}
```

Fig. 10.10 **`Boss`** class inherits from class **`Employee`**. (Part 2 of 2.)

Class **`CommissionWorker`** (Fig. 10.11) also inherits from class **`Employee`**. The constructor for this class (lines 12–20) receives as arguments a first name, a last name, a salary, a commission and a quantity of items sold. Line 15 passes the first name and last name to the base-class **`Employee`** constructor. Class **`CommissionWorker`** also provides properties **`WeeklySalary`** (lines 23–36), **`Commission`** (lines 39–52) and **`Quantity`** (lines 55–68); method **`Earnings`** (lines 72–75), which calculates the worker's wages; and method **`ToString`** (lines 78–81), which returns a string that indicates the employee type (i.e., **`"CommissionWorker: "`**) and the worker's name.

```
 1  // Fig. 10.11: CommisionWorker.cs
 2  // CommissionWorker class derived from Employee
 3  using System;
 4
 5  public class CommissionWorker : Employee
 6  {
 7     private decimal salary;     // base weekly salary
 8     private decimal commission; // amount paid per item sold
 9     private int quantity;       // total items sold
10
```

Fig. 10.11 **`CommissionWorker`** class inherits from class **`Employee`**. (Part 1 of 3.)

```
   // constructor
   public CommissionWorker( string firstNameValue,
      string lastNameValue, decimal salaryValue,
      decimal commissionValue, int quantityValue )
      : base( firstNameValue, lastNameValue )
   {
      WeeklySalary = salaryValue;
      Commission = commissionValue;
      Quantity = quantityValue;
   }

   // property WeeklySalary
   public decimal WeeklySalary
   {
      get
      {
         return salary;
      }

      set
      {
         // ensure non-negative salary value
         if ( value > 0 )
            salary = value;
      }
   }

   // property Commission
   public decimal Commission
   {
      get
      {
         return commission;
      }

      set
      {
         // ensure non-negative commission value
         if ( value > 0 )
            commission = value;
      }
   }

   // property Quantity
   public int Quantity
   {
      get
      {
         return quantity;
      }

```

Fig. 10.11 `CommissionWorker` class inherits from class `Employee`. (Part 2 of 3.)

```
      set
      {
         // ensure non-negative quantity value
         if ( value > 0 )
            quantity = value;
      }
   }

   // override base-class method to calculate
   // CommissionWorker's earnings
   public override decimal Earnings()
   {
      return WeeklySalary + Commission * Quantity;
   }

   // return string representation of CommissionWorker
   public override string ToString()
   {
      return "CommissionWorker: " + base.ToString();
   }

} // end class CommissionWorker
```

Fig. 10.11 `CommissionWorker` class inherits from class `Employee`. (Part 3 of 3.)

Class **`PieceWorker`** (Fig. 10.12) inherits from class **`Employee`**. The constructor for this class (lines 11–18) receives as arguments a first name, a last name, a wage per piece and a quantity of items produced. Line 14 then passes the first name and last name to the base-class **`Employee`** constructor. Class **`PieceWorker`** also provides properties **`WagePerPiece`** (lines 21–33) and **`Quantity`** (lines 36–48); method **`Earnings`** (lines 52–55), which calculates a piece worker's earnings; and method **`ToString`** (lines 58–61), which returns a string that indicates the type of the employee (i.e., **`"PieceWorker: "`**) and the piece worker's name.

```
1  // Fig. 10.12: PieceWorker.cs
2  // PieceWorker class derived from Employee.
3  using System;
4
5  public class PieceWorker : Employee
6  {
7     private decimal wagePerPiece; // wage per piece produced
8     private int quantity;         // quantity of pieces produced
9
10    // constructor
11    public PieceWorker( string firstNameValue,
12       string lastNameValue, decimal wagePerPieceValue,
13       int quantityValue )
14       : base( firstNameValue, lastNameValue )
15    {
16       WagePerPiece = wagePerPieceValue;
```

Fig. 10.12 `PieceWorker` class inherits from class `Employee`. (Part 1 of 2.)

```
      Quantity = quantityValue;
   }

   // property WagePerPiece
   public decimal WagePerPiece
   {
      get
      {
         return wagePerPiece;
      }

      set
      {
         if ( value > 0 )
            wagePerPiece = value;
      }
   }

   // property Quantity
   public int Quantity
   {
      get
      {
         return quantity;
      }

      set
      {
         if ( value > 0 )
            quantity = value;
      }
   }

   // override base-class method to calculate
   // PieceWorker's earnings
   public override decimal Earnings()
   {
      return Quantity * WagePerPiece;
   }

   // return string representation of PieceWorker
   public override string ToString()
   {
      return "PieceWorker: " + base.ToString();
   }
}
```

Fig. 10.12 `PieceWorker` class inherits from class `Employee`. (Part 2 of 2.)

Class **`HourlyWorker`** (Fig. 10.13) inherits from class **`Employee`**. The constructor for this class (lines 11–17) receives as arguments a first name, a last name, a wage and the number of hours worked. Line 13 passes the first name and last name to the base-class

`Employee` constructor. Class `HourlyWorker` also provides properties `Wage` (lines 20–33) and `HoursWorked` (lines 36–49); method `Earnings` (lines 53–70), which calculates an hourly worker's earnings; and method `ToString` (lines 73–76), which returns a string that indicates the type of the employee (i.e., `"HourlyWorker:"`) and the hourly worker's name. Note that hourly workers are paid "time-and-a-half" for "overtime" (i.e., hours worked in excess of 40 hours).

```
1  // Fig. 10.13: HourlyWorker.cs
2  // HourlyWorker class derived from Employee.
3  using System;
4
5  public class HourlyWorker : Employee
6  {
7     private decimal wage;       // wage per hour of work
8     private double hoursWorked; // hours worked during week
9
10    // constructor
11    public HourlyWorker( string firstNameValue, string LastNameValue,
12       decimal wageValue, double hoursWorkedValue )
13       : base( firstNameValue, LastNameValue )
14    {
15       Wage = wageValue;
16       HoursWorked = hoursWorkedValue;
17    }
18
19    // property Wage
20    public decimal Wage
21    {
22       get
23       {
24          return wage;
25       }
26
27       set
28       {
29          // ensure non-negative wage value
30          if ( value > 0 )
31             wage = value;
32       }
33    }
34
35    // property HoursWorked
36    public double HoursWorked
37    {
38       get
39       {
40          return hoursWorked;
41       }
42
```

Fig. 10.13 `HourlyWorker` class inherits from class `Employee` (Part 1 of 2.).

```
      set
      {
         // ensure non-negative hoursWorked value
         if ( value > 0 )
            hoursWorked = value;
      }
   }

   // override base-class method to calculate
   // HourlyWorker earnings
   public override decimal Earnings()
   {
      // compensate for overtime (paid "time-and-a-half")
      if ( HoursWorked <= 40 )
      {
         return Wage * Convert.ToDecimal( HoursWorked );
      }

      else
      {
         // calculate base and overtime pay
         decimal basePay = Wage * Convert.ToDecimal( 40 );
         decimal overtimePay = Wage * 1.5M *
            Convert.ToDecimal( HoursWorked - 40 );

         return basePay + overtimePay;
      }
   }

   // return string representation of HourlyWorker
   public override string ToString()
   {
      return "HourlyWorker: " + base.ToString();
   }
}
```

Fig. 10.13 **`HourlyWorker`** class inherits from class **`Employee`** (Part 2 of 2.).

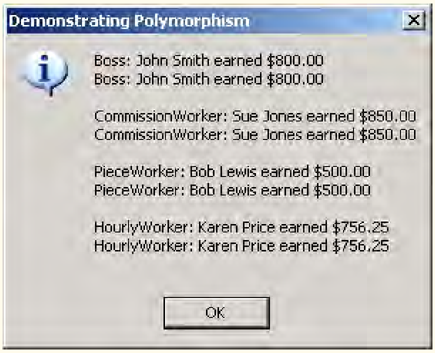
Method **`Main`** (lines 9–48) of class **`EmployeesTest`** (Fig. 10.14) declares **`Employee`** reference **`employee`** (line 22). Each employee type is handled similarly in **`Main`**, so we discuss only the manipulations of the **`Boss`** object.

Line 11 assigns to **`Boss`** reference **`boss`** a **`Boss`** object and passes to its constructor the boss's first name ("**`John`**"), last name ("**`Smith`**") and fixed weekly salary (**`800`**). Line 22 assigns the derived-class reference **`boss`** to the base-class **`Employee`** reference **`employee`**, so we can demonstrate the polymorphic determination of **`boss`**'s earnings. Line 24 passes reference **`employee`** as an argument to method **`GetString`** (lines 51–55), which polymorphically invokes methods **`ToString`** and **`Earnings`** on the **`Employee`** object the method receives as an argument. At this point, C# determines that the object passed to **`GetString`** is of type **`Boss`**, so lines 53–54 invoke **`Boss`** methods **`ToString`** and **`Earnings`**. These are classic examples of polymorphic behavior.

```
// Fig. 10.14: EmployeesTest.cs
// Demonstrates polymorphism by displaying earnings
// for various Employee types.
using System;
using System.Windows.Forms;

public class EmployeesTest
{
   public static void Main( string[] args )
   {
      Boss boss = new Boss( "John", "Smith", 800 );

      CommissionWorker commissionWorker =
         new CommissionWorker( "Sue", "Jones", 400, 3, 150 );

      PieceWorker pieceWorker = new PieceWorker( "Bob", "Lewis",
         Convert.ToDecimal( 2.5 ), 200 );

      HourlyWorker hourlyWorker = new HourlyWorker( "Karen",
         "Price", Convert.ToDecimal( 13.75 ), 50 );

      Employee employee = boss;

      string output = GetString( employee ) + boss + " earned " +
           boss.Earnings().ToString( "C" ) + "\n\n";

      employee = commissionWorker;

      output += GetString( employee ) + commissionWorker +
         " earned " +
         commissionWorker.Earnings().ToString( "C" ) + "\n\n";

      employee = pieceWorker;

      output += GetString( employee ) + pieceWorker +
         " earned " + pieceWorker.Earnings().ToString( "C" ) +
         "\n\n";

      employee = hourlyWorker;

      output += GetString( employee ) + hourlyWorker +
         " earned " + hourlyWorker.Earnings().ToString( "C" ) +
         "\n\n";

      MessageBox.Show( output, "Demonstrating Polymorphism",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method Main

   // return string that contains Employee information
   public static string GetString( Employee worker )
   {
```

Fig. 10.14 **`EmployeesTest`** class tests the **`Employee`** class hierarchy. (Part 1 of 2.)

```
53          return worker.ToString() + " earned " +
54             worker.Earnings().ToString( "C" ) + "\n";
55       }
56    }
```



```
Boss: John Smith earned $800.00
Boss: John Smith earned $800.00

CommissionWorker: Sue Jones earned $850.00
CommissionWorker: Sue Jones earned $850.00

PieceWorker: Bob Lewis earned $500.00
PieceWorker: Bob Lewis earned $500.00

HourlyWorker: Karen Price earned $756.25
HourlyWorker: Karen Price earned $756.25
```

Fig. 10.14 **`EmployeesTest`** class tests the **`Employee`** class hierarchy. (Part 2 of 2.)

Method **`Earnings`** returns a **`Decimal`** object on which line 54 then calls method **`ToString`**. In this case, the string **`"C"`**, which is passed to an overloaded version of **`Decimal`** method **`ToString`**, stands for **`Currency`**, and **`ToString`** formats the string as a currency amount.

When method **`GetString`** returns to **`Main`**, lines 24–25 explicitly invoke methods **`ToString`** and **`Earnings`** through derived-class **`Boss`** reference **`boss`** to show the method invocations that do not use polymorphic processing. The output generated in lines 24–25 is identical to that generated by methods **`ToString`** and **`Earnings`** through base-class reference **`employee`** (i.e., the methods that use polymorphism), which verifies that the polymorphic methods invoke the appropriate methods in derived class **`Boss`**.

To prove that the base-class reference **`employee`** can invoke the proper derived-class versions of methods **`ToString`** and **`Earnings`** for the other types of employees, lines 27, 33 and 39 assign to base-class reference **`employee`** a different type of **`Employee`** object (**`CommissionWorker`**, **`PieceWorker`** and **`HourlyWorker`**, respectively). After each assignment, the application calls method **`GetString`** to return the results via the base-class reference. Then, the application calls methods **`ToString`** and **`Earnings`** of each derived-class reference to show that C# correctly associates each method call to its corresponding derived-class object.

10.9 Case Study: Creating and Using Interfaces

We now present two more examples of polymorphism using *interfaces* that specify sets of **`public`** services (i.e., methods and properties) that classes must implement. An interface is used when there is no default implementation to inherit (i.e., no instance variables and no default-method implementations). Whereas an abstract class is best used for providing data and services for objects in a hierarchical relationship, an interface can be used for providing services that “bring together” disparate objects that relate to one another only through that interface.

An interface definition begins with the keyword ***`interface`*** and contains a list of `public` methods and properties. To use an interface, a class must specify that it implements the interface and must provide implementations for every method and property specified in the interface definition. A class that implements an interface effectively signs a contract with the compiler that states, “this class will define all the methods and properties specified by the interface.”

Common Programming Error 10.8

When a class implements an `interface`*, leaving even a single* `interface` *method or property undefined is an error. The class must define every method and property in the* `interface`*.*

Common Programming Error 10.9

In C#, an `interface` *can be declared only as* `public`*; the declaration of an* `interface` *as* `private` *or* `protected` *is an error.*

Interfaces provide uniform sets of methods and properties for objects of disparate classes. These methods and properties enable programs to process the objects of those disparate classes polymorphically. For example, consider disparate objects that represent a person, a tree, a car and a file. These objects have “nothing to do” with one another—a person has a first name and last name; a tree has a trunk, a set of branches and a bunch of leaves; a car has wheels, gears and several other mechanisms that enable the car to move; and a file contains data. Because of the lack of commonality among these classes, modeling them via an inheritance hierarchy with a common base class seems illogical. However, these objects certainly have at least one common characteristic—an age. A person’s age is represented by the number of years since that person was born; a tree’s age is represented by the number of rings in its trunk; a car’s age is represented by its manufacture date; and file’s age is represented by its creation date. We can use an interface that provides a method or property that objects of these disparate classes can implement to return each object’s age.

In this example, we use interface `IAge` (Fig. 10.15) to return the age information for classes `Person` (Fig. 10.16) and `Tree` (Fig. 10.17). The definition of interface `IAge` begins at line 4 with `public interface` and ends at line 8 with a closing curly brace. Lines 6–7 specify read-only properties `Age` and `Name`, for which every class that implements interface `IAge` must provide implementations. Declaring these properties as read-only is not required—an interface can also provide methods, write-only properties and properties with both *get* and *set* accessors. By containing these property declarations, interface `IAge` provides an opportunity for an object that implements `IAge` to return its age and name, respectively. However, the classes that implement these methods are not “required” by either interface `IAge` or C# to return an age and a name. The compiler requires only that classes implementing interface `IAge` provide implementations for the interface’s properties.

Line 5 of Fig. 10.16 use C#’s inheritance notation (i.e., *ClassName* `:` *InterfaceName*) to indicate that class `Person` implements interface `IAge`. In this example, class `Person` implements only one interface. A class can implement any number of interfaces in addition to inheriting from one class. To implement more than one interface, the class definition must provide a comma-separated list of interface names after the semicolon. Class `Person` has member variables `yearBorn`, `firstName` and `lastName` (lines 7–9), for which the constructor (lines 12–22) sets values. Because class `Person` implements interface `IAge`, class `Person` must implement properties `Age` and `Name`—defined on lines

25–31 and lines 34–40, respectively. Property **`Age`** allows the client to obtain the person's age, and property **`Name`** returns a **`String`** containing **`firstName`** and **`lastName`**. Note that property **`Age`** calculates the person's age by subtracting **`yearBorn`** from the current year (via property **`Year`** of property **`DateTime.Now`**, which returns the current date). These properties satisfy the implementation requirements defined in interface **`IAge`**, so class **`Person`** has fulfilled its "contract" with the compiler.

```
// Fig. 10.15: IAge.cs
// Interface IAge declares property for setting and getting age.

public interface IAge
{
   int Age { get; }
   string Name { get; }
}
```

Fig. 10.15 Interface for returning age of objects of disparate classes.

```
 1  // Fig. 10.16: Person.cs
 2  // Class Person has a birthday.
 3  using System;
 4  
 5  public class Person : IAge
 6  {
 7     private string firstName;
 8     private string lastName;
 9     private int yearBorn;
10  
11     // constructor
12     public Person( string firstNameValue, string lastNameValue,
13        int yearBornValue )
14     {
15        firstName = firstNameValue;
16        lastName = lastNameValue;
17  
18       if ( yearBornValue > 0 && yearBornValue <= DateTime.Now.Year )
19           yearBorn = yearBornValue;
20        else
21           yearBorn = DateTime.Now.Year;
22     }
23  
24     // property Age implementation of interface IAge
25     public int Age
26     {
27        get
28        {
29           return DateTime.Now.Year - yearBorn;
30        }
31     }
32  
```

Fig. 10.16 **`Person`** class implements **`IAge`** interface. (Part 1 of 2.)

```
33    // property Name implementation of interface IAge
34    public string Name
35    {
36       get
37       {
38          return firstName + " " + lastName;
39       }
40    }
41
42 } // end class Person
```

Fig. 10.16 **`Person`** class implements **`IAge`** interface. (Part 2 of 2.)

Class **`Tree`** (Fig. 10.17) also implements interface **`IAge`**. Class **`Tree`** has member variables **`rings`** (line 7), which represents the number of rings inside the tree's trunk—this variable corresponds directly to the tree's age. The **`Tree`** constructor (lines 10–14) receives as an argument an **`int`** that specifies in which year the tree was planted. Class **`Tree`** includes method **`AddRing`** (lines 17–20), which enables a program to increment the number of rings in the tree. Because class **`Tree`** implements interface **`IAge`**, class **`Tree`** must implement properties **`Age`** and **`Name`**—defined on lines 23–29 and lines 32–38, respectively. Property **`Age`** returns the value of **`rings`**, and property **`Name`** returns **`string`** "**`Tree`**."

```
 1 // Fig. 10.17: Tree.cs
 2 // Class Tree contains number of rings corresponding to its age.
 3 using System;
 4
 5 public class Tree : IAge
 6 {
 7    private int rings; // number of rings in tree trunk
 8
 9    // constructor
10    public Tree( int yearPlanted )
11    {
12       // count number of rings in Tree
13       rings = DateTime.Now.Year - yearPlanted;
14    }
15
16    // increment rings
17    public void AddRing()
18    {
19       rings++;
20    }
21
22    // property Age implementation of interface IAge
23    public int Age
24    {
25       get
26       {
```

Fig. 10.17 **`Tree`** class implements **`IAge`** interface. (Part 1 of 2.)

```
            return rings;
         }
      }

      // property Name implementation of interface IAge
      public string Name
      {
         get
         {
            return "Tree";
         }
      }

   } // end class Tree
```

Fig. 10.17 `Tree` class implements `IAge` interface. (Part 2 of 2.)

Class **`InterfacesTest`** (Fig. 10.18) demonstrates polymorphism on the objects of disparate classes **`Person`** and **`Tree`**. Line 9 instantiates object **`tree`** of class **`Tree`**, and line 10 instantiates object **`person`** of class **`Person`**. Line 13 declares **`iAgeArray`**—an array of two references to **`IAge`** objects. Line 16 and 19 assign **`tree`** and **`person`** to the first and second reference in **`iAgeArray`**, respectively. Lines 22–23 invoke method **`ToString`** on **`tree`**, then invoke its properties **`Age`** and **`Name`** to return age and name information for object **`tree`**. Lines 26–27 invoke method **`ToString`** on **`person`**, then invoke its properties **`Age`** and **`Name`** to return age and name information for object **`person`**. Next, we manipulate these objects polymorphically through the **`iAgeArray`** of references to **`IAge`** objects. Lines 30–34 define a **`foreach`** structure that uses properties **`Age`** and **`Name`** to obtain age and name information for each **`IAge`** object in **`iAgeArray`**. Note that a program also can invoke class **`Object`**'s **`public`** methods (e.g., **`ToString`**) using any interface reference. This is possible because every object inherits directly or indirectly from class **`Object`**. Therefore, every object is guaranteed to have the class **`Object`**'s **`public`** methods.

Software Engineering Observation 10.11

In C#, an interface reference may invoke methods and properties that the interface declares and the **`public`** *methods of class* **`Object`***.*

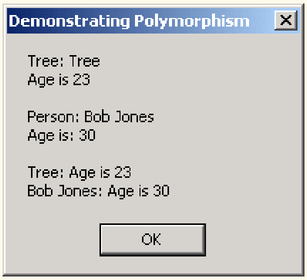
Software Engineering Observation 10.12

In C#, an interface provides only those **`public`** *services declared in the interface, whereas an abstract class provides the* **`public`** *services defined in the abstract class and those members inherited from the abstract class's base class.*

```
1  // Fig. 10.18: InterfacesTest.cs
2  // Demonstrating polymorphism with interfaces.
3  using System.Windows.Forms;
4
5  public class InterfacesTest
6  {
```

Fig. 10.18 Demonstrate polymorphism on objects of disparate classes. (Part 1 of 2.)

```
   public static void Main( string[] args )
   {
      Tree tree = new Tree( 1978 );
      Person person = new Person( "Bob", "Jones", 1971 );

      // create array of IAge references
      IAge[] iAgeArray = new IAge[ 2 ];

      // iAgeArray[ 0 ] refers to Tree object polymorphically
      iAgeArray[ 0 ] = tree;

      // iAgeArray[ 1 ] refers to Person object polymorphically
      iAgeArray[ 1 ] = person;

      // display tree information
      string output = tree + ": " + tree.Name + "\nAge is " +
         tree.Age + "\n\n";

      // display person information
      output += person + ": " + person.Name + "\nAge is: "
         + person.Age + "\n\n";

      // display name and age for each IAge object in iAgeArray
      foreach ( IAge ageReference in iAgeArray )
      {
         output += ageReference.Name + ": Age is " +
            ageReference.Age + "\n";
      }

      MessageBox.Show( output, "Demonstrating Polymorphism" );

   } // end method Main

} // end class InterfacesTest
```



```
Tree: Tree
Age is 23

Person: Bob Jones
Age is: 30

Tree: Age is 23
Bob Jones: Age is 30
```

Fig. 10.18 Demonstrate polymorphism on objects of disparate classes. (Part 2 of 2.)

Our next example reexamines the **`Point`**–**`Circle`**–**`Cylinder`** hierarchy using an interface, rather than using an abstract class, to describe the common methods and properties of the classes in the hierarchy. We now show how a class can implement an interface, then act as a base class for derived classes to inherit the implementation. We create interface **`IShape`** (Fig. 10.19), which specifies methods **`Area`** and **`Volume`** and property **`Name`** (lines 8–10). Every class that implements interface **`IShape`** must provide implementations for these two methods and this read-only property. Note that, even though the methods in this example interface do not receive arguments, interface methods can receive arguments (just as regular methods can).

Good Programming Practice 10.1

*By convention, begin the name of each interface with "***I***."*

Because class **Point3** (Fig. 10.20) implements interface **IShape**, class **Point3** must implement all three **IShape** members. Lines 59–62 implement method **Area**, which returns **0**, because points have an area of zero. Lines 65–68 implement method **Volume**, which also returns **0**, because points have a volume of zero. Lines 71–77 implement read-only property **Name**, which returns the class name as a **string** (**"Point3"**).

When a class implements an interface, the class enters the same kind of *is-a* relationship that inheritance establishes. In our example, class **Point3** implements interface **IShape**. Therefore, a **Point3** object *is an* **IShape**, and objects of any class that inherits from **Point3** are also **IShape**s. For example, class **Circle3** (Fig. 10.21) inherits from class **Point3**; thus, a **Circle3** *is an* **IShape**. Class **Circle3** implements interface **IShape** implicitly and inherits the **IShape** methods that class **Point** implemented. Because circles do not have volume, class **Circle3** does not override class **Point3**'s **Volume** method, which returns zero. However, we do not want to use the class **Point3** method **Area** or property **Name** for class **Circle3**. Class **Circle3** should provide its own implementation for these, because the area and name of a circle differ from those of a point. Lines 52–55 override method **Area** to return the circle's area, and lines 65–71 override property **Name** to return **String "Circle3"**.

Class **Cylinder3** (Fig. 10.22) inherits from class **Circle3**. **Cylinder3** implements interface **IShape** implicitly, because **Cylinder3** derives from **Point3**, which implements interface **IShape**. **Cylinder3** inherits method **Area** and property **Name** from **Circle3** and method **Volume** from **Point3**. However, **Cylinder3** overrides property **Name** and methods **Area** and **Volume** to perform **Cylinder3**-specific operations. Lines 40–43 override method **Area** to return the cylinder's surface area, lines 46–49 override method **Volume** to return the cylinder's volume and lines 59–65 override property **Name** to return **String "Cylinder3"**. Note that class **Point3** marks these methods/properties as **virtual**, enabling derived classes to override them.

```
// Fig. 10.19: IShape.cs
// Interface IShape for Point, Circle, Cylinder Hierarchy.

public interface IShape
{
   // classes that implement IShape must implement these methods
   // and this property
   double Area();
   double Volume();
   string Name { get; }
}
```

Fig. 10.19 **IShape** interface provides methods **Area** and **Volume** and property **Name**.

```
// Fig. 10.20: Point3.cs
// Point3 implements interface IShape and represents
// an x-y coordinate pair.
using System;

// Point3 implements IShape
public class Point3 : IShape
{
   private int x, y; // Point3 coordinates

   // default constructor
   public Point3()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point3( int xValue, int yValue )
   {
      X = xValue;
      Y = yValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

      set
      {
         x = value;
      }
   }

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value;
      }
   }

```

Fig. 10.20 `Point3` class implements interface `IShape`. (Part 1 of 2.)

```
   // return string representation of Point3 object
   public override string ToString()
   {
      return "[" + X + ", " + Y + "]";
   }

   // implement interface IShape method Area
   public virtual double Area()
   {
      return 0;
   }

   // implement interface IShape method Volume
   public virtual double Volume()
   {
      return 0;
   }

   // implement property Name of IShape
   public virtual string Name
   {
      get
      {
         return "Point3";
      }
   }

} // end class Point3
```

Fig. 10.20 **`Point3`** class implements interface **`IShape`**. (Part 2 of 2.)

```
// Fig. 10.21: Circle3.cs
// Circle3 inherits from class Point3 and overrides key members.
using System;

// Circle3 inherits from class Point3
public class Circle3 : Point3
{
   private double radius; // Circle3 radius

   // default constructor
   public Circle3()
   {
      // implicit call to Point3 constructor occurs here
   }

   // constructor
   public Circle3( int xValue, int yValue, double radiusValue )
      : base( xValue, yValue )
   {
      Radius = radiusValue;
   }
```

Fig. 10.21 **`Circle3`** class inherits from class **`Point3`**. (Part 1 of 2.)

```

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

      set
      {
         // ensure non-negative Radius value
         if ( value >= 0 )
            radius = value;
      }
   }

   // calculate Circle3 diameter
   public double Diameter()
   {
      return Radius * 2;
   }

   // calculate Circle3 circumference
   public double Circumference()
   {
      return Math.PI * Diameter();
   }

   // calculate Circle3 area
   public override double Area()
   {
      return Math.PI * Math.Pow( Radius, 2 );
   }

   // return string representation of Circle3 object
   public override string ToString()
   {
      return "Center = " + base.ToString() +
         "; Radius = " + Radius;
   }

   // override property Name from class Point3
   public override string Name
   {
      get
      {
         return "Circle3";
      }
   }

} // end class Circle3
```

Fig. 10.21 `Circle3` class inherits from class `Point3`. (Part 2 of 2.)

```
// Fig. 10.22: Cylinder3.cs
// Cylinder3 inherits from class Circle2 and overrides key members.
using System;

// Cylinder3 inherits from class Circle3
public class Cylinder3 : Circle3
{
   private double height; // Cylinder3 height

   // default constructor
   public Cylinder3()
   {
      // implicit call to Circle3 constructor occurs here
   }

   // constructor
   public Cylinder3( int xValue, int yValue, double radiusValue,
      double heightValue ) : base( xValue, yValue, radiusValue )
   {
      Height = heightValue;
   }

   // property Height
   public double Height
   {
      get
      {
         return height;
      }

      set
      {
         // ensure non-negative Height value
         if ( value >= 0 )
            height = value;
      }
   }

   // calculate Cylinder3 area
   public override double Area()
   {
      return 2 * base.Area() + base.Circumference() * Height;
   }

   // calculate Cylinder3 volume
   public override double Volume()
   {
      return base.Area() * Height;
   }

   // return string representation of Cylinder3 object
   public override string ToString()
   {
```

Fig. 10.22 `Cylinder3` class inherits from class `Circle3`. (Part 1 of 2.)

```
      return "Center = " + base.ToString() +
         "; Height = " + Height;
   }

   // override property Name from class Circle3
   public override string Name
   {
      get
      {
         return "Cylinder3";
      }
   }

} // end class Cylinder3
```

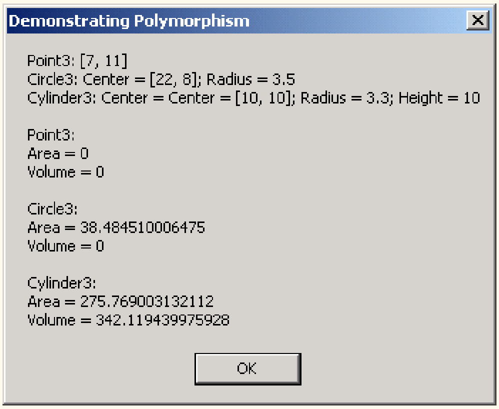
Fig. 10.22 `Cylinder3` class inherits from class `Circle3`. (Part 2 of 2.)

Class **`Interfaces2Test`** (Fig. 10.23) demonstrates our point-circle-cylinder hierarchy that uses interfaces. Class **`Interfaces2Test`** has only two differences from the example in Fig. 10.8, which tested the class hierarchy created from the **`abstract`** base class **`Shape`**. In Fig. 10.23, line 17 declares **`arrayOfShapes`** as an array of **`IShape`** interface references, rather than **`Shape`** base-class references.

```
1  // Fig. 10.23: Interfaces2Test.cs
2  // Demonstrating polymorphism with interfaces in
3  // Point-Circle-Cylinder hierarchy.
4
5  using System.Windows.Forms;
6
7  public class Interfaces2Test
8  {
9     public static void Main( string[] args )
10    {
11       // instantiate Point3, Circle3 and Cylinder3 objects
12       Point3 point = new Point3( 7, 11 );
13       Circle3 circle = new Circle3( 22, 8, 3.5 );
14       Cylinder3 cylinder = new Cylinder3( 10, 10, 3.3, 10 );
15
16       // create array of IShape references
17       IShape[] arrayOfShapes = new IShape[ 3 ];
18
19       // arrayOfShapes[ 0 ] references Point3 object
20       arrayOfShapes[ 0 ] = point;
21
22       // arrayOfShapes[ 1 ] references Circle3 object
23       arrayOfShapes[ 1 ] = circle;
24
25       // arrayOfShapes[ 2 ] references Cylinder3 object
26       arrayOfShapes[ 2 ] = cylinder;
27
```

Fig. 10.23 `Interfaces2Test` uses interfaces to demonstrate polymorphism in Point-Circle-Cylinder hierarchy (Part 1 of 2.).

```
      string output = point.Name + ": " + point + "\n" +
         circle.Name + ": " + circle + "\n" +
         cylinder.Name + ": " + cylinder;

      foreach ( IShape shape in arrayOfShapes )
      {
         output += "\n\n" + shape.Name + ":\nArea = " +
            shape.Area() + "\nVolume = " + shape.Volume();
      }

      MessageBox.Show( output, "Demonstrating Polymorphism" );
   }
}
```



```
Point3: [7, 11]
Circle3: Center = [22, 8]; Radius = 3.5
Cylinder3: Center = Center = [10, 10]; Radius = 3.3; Height = 10

Point3:
Area = 0
Volume = 0

Circle3:
Area = 38.484510006475
Volume = 0

Cylinder3:
Area = 275.769003132112
Volume = 342.119439975928
```

Fig. 10.23 `Interfaces2Test` uses interfaces to demonstrate polymorphism in Point-Circle-Cylinder hierarchy (Part 2 of 2.).

10.10 Delegates

In Chapter 6, we discussed how objects can pass member variables as arguments to methods. However, sometimes, it is beneficial for objects to pass methods as arguments to other methods. For example, suppose that you wish to sort a series of values in ascending and descending order. Rather than provide separate ascending and descending sorting methods (one for each type of comparison), we could provide a single method that receives as an argument a reference to the comparison method to use. To perform an ascending sort, we could pass to the sorting method the reference to the ascending-sort-comparison method; to perform a descending sort, we could pass to the sorting method the reference to the descending-sort-comparison method. The sorting method then would use this reference to sort the list—the sorting method would not need to know whether it is performing an ascending or descending sort.

C# does not allow the passing of method references directly as arguments to other methods, but does provide *delegates*, which are classes that encapsulate sets of references to methods. A delegate object that contains method references can be passed to another method. Rather than send a method reference directly, an object can send the delegate instance, which contains the reference of the method that we would like to send. The method that receives the reference to the delegate then can invoke the methods the delegate contains.

A delegate that contains a single method is known as a *singlecast delegate* and is created or derived from class **`Delegate`**. Delegates that contain multiple methods are *multicast delegates* and are created or derived from class **`MulticastDelegate`**. Both delegate classes belong to namespace **`System`**.

To use a delegate, we first must declare one. The delegate's declaration specifies a method header (parameters and return value). Methods whose references will be contained within a delegate object must have the same method header as that defined in the delegate declaration. We then create methods that have this signature. The second step is to create a delegate instance that contains a reference to that method. After we create the delegate instance, we can invoke the method reference that it contains. We show this process in our next example.

Class **`DelegateBubbleSort`** (Fig. 10.24), which is a modified version of the bubble-sort example in Chapter 7, uses delegates to sort an integer array in ascending or descending order. Lines 6–7 provide the declaration for delegate **`Comparator`**. To declare a delegate (line 7), we declare a signature of a method—keyword **`delegate`** after the member-access modifier (in this case, **`public`**), followed by the return type, the delegate name and parameter list. Delegate **`Comparator`** defines a method signature for methods that receive two **`int`** arguments and return a **`bool`**. Note that delegate **`Comparator`** contains no body. As we soon demonstrate, our application (Fig. 10.25) implements methods that adhere to delegate **`Comparator`**'s signature, then passes these methods (as arguments of type **`Comparator`**) to method **`SortArray`**. The declaration of a delegate does not define its intended role or implementation; our application uses this particular delegate when comparing two **`int`**s, but other applications might use it for different purposes.

```
 1  // Fig. 10.24: DelegateBubbleSort.cs
 2  // Demonstrating delegates for sorting numbers.
 3
 4  public class DelegateBubbleSort
 5  {
 6     public delegate bool Comparator( int element1,
 7        int element2 );
 8
 9     // sort array using Comparator delegate
10     public static void SortArray( int[] array,
11        Comparator Compare )
12     {
13        for ( int pass = 0; pass < array.Length; pass++ )
14
15           for ( int i = 0; i < array.Length - 1; i++ )
16
17              if ( Compare( array[ i ], array [ i + 1 ] ) )
18                 Swap( ref array[ i ], ref array[ i + 1 ] );
19     }
20
21     // swap two elements
22     private static void Swap( ref int firstElement,
23        ref int secondElement )
24     {
```

Fig. 10.24 Bubble sort using delegates. (Part 1 of 2.)

```
25       int hold = firstElement;
26       firstElement = secondElement;
27       secondElement = hold;
28    }
29 }
```

Fig. 10.24 Bubble sort using delegates. (Part 2 of 2.)

Lines 10–19 define method **SortArray**, which takes an array and a reference to a **Comparator** delegate object as arguments. Method **SortArray** modifies the array by sorting its contents. Line 17 uses the delegate method to determine how to sort the array. Line 17 invokes the method enclosed within the delegate object by treating the delegate reference as the method that the delegate object contains. C# invokes the enclosed method reference directly, passing it parameters **array[i]** and **array[i + 1]**. The **Comparator** determines the sorting order for its two arguments. If the **Comparator** returns **true**, the two elements are out of order, so line 18 invokes method **Swap** (lines 22–28) to swap the elements. If the **Comparator** returns **false**, the two elements are in the correct order. To sort in ascending order, the **Comparator** returns **true** when the first element being compared is greater than the second element being compared. Similarly, to sort in descending order, the **Comparator** returns **true** when the first element being compared is less than the second element being compared.

Class **BubbleSortForm** (Fig. 10.25) displays a **Form** with two text boxes and three buttons. The first text box displays a list of unsorted numbers, and the second box displays the same list of numbers after they are sorted. The **Create Data** button creates the list of unsorted values. The **Sort Ascending** and **Sort Descending** buttons sort the array in ascending and descending order, respectively. Methods **SortAscending** (lines 42–45) and **SortDescending** (lines 60–63) each have a signature that corresponds with the signature defined by the **Comparator** delegate declaration (i.e., each receives two **int**s and returns a **bool**). As we will see, the program passes to **DelegateBubbleSort** method **SortArray** delegates containing references to methods **SortAscending** and **SortDescending**, which will specify class **DelegateBubbleSort**'s sorting behavior.

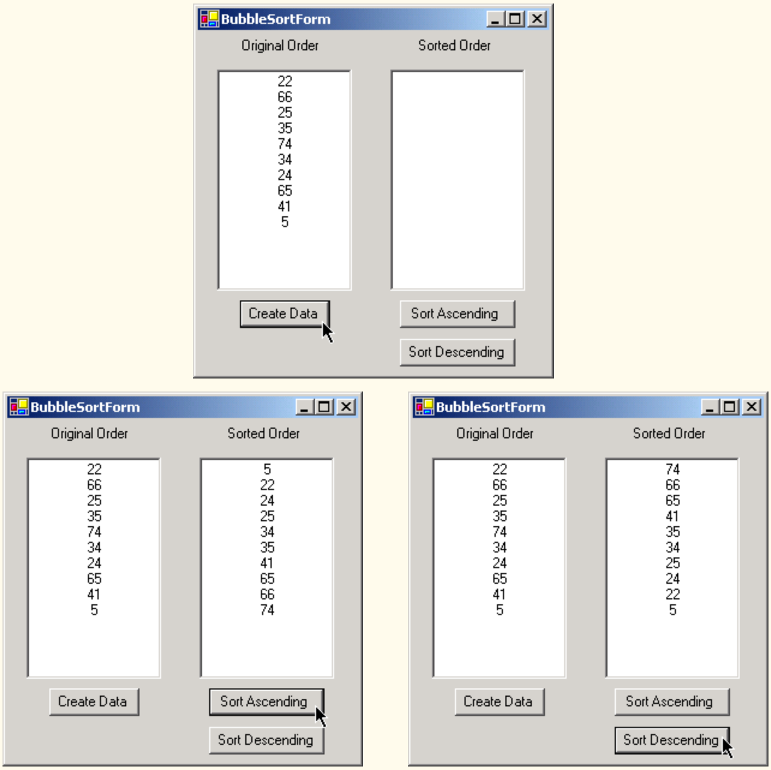
```
 1 // Fig. 10.25: BubbleSortForm.cs
 2 // Demonstrates bubble sort using delegates to determine
 3 // the sort order.
 4 using System;
 5 using System.Drawing;
 6 using System.Collections;
 7 using System.ComponentModel;
 8 using System.Windows.Forms;
 9
10 public class BubbleSortForm : System.Windows.Forms.Form
11 {
12    private System.Windows.Forms.TextBox originalTextBox;
13    private System.Windows.Forms.TextBox sortedTextBox;
14    private System.Windows.Forms.Button createButton;
15    private System.Windows.Forms.Button ascendingButton;
```

Fig. 10.25 Bubble-sort **Form** application. (Part 1 of 3.)

```
   private System.Windows.Forms.Button descendingButton;
   private System.Windows.Forms.Label originalLabel;
   private System.Windows.Forms.Label sortedLabel;

   private int[] elementArray = new int[ 10 ];

   // create randomly generated set of numbers to sort
   private void createButton_Click( object sender,
      System.EventArgs e )
   {
      // clear TextBoxes
      originalTextBox.Clear();
      sortedTextBox.Clear();

      // create random-number generator
      Random randomNumber = new Random();

      // populate elementArray with random integers
      for ( int i = 0; i < elementArray.Length; i++ )
      {
         elementArray[ i ] = randomNumber.Next( 100 );
         originalTextBox.Text += elementArray[ i ] + "\r\n";
      }
   }

   // delegate implementation for ascending sort
   private bool SortAscending( int element1, int element2 )
   {
      return element1 > element2;
   }

   // sort randomly generated numbers in ascending order
   private void ascendingButton_Click( object sender,
      System.EventArgs e )
   {
      // sort array, passing delegate for SortAscending
      DelegateBubbleSort.SortArray( elementArray,
         new DelegateBubbleSort.Comparator(
            SortAscending ) );

      DisplayResults();
   }

   // delegate implementation for descending sort
   private bool SortDescending( int element1, int element2 )
   {
      return element1 < element2;
   }

   // sort randomly generating numbers in descending order
   private void descendingButton_Click( object sender,
      System.EventArgs e )
   {
```

Fig. 10.25 Bubble-sort **`Form`** application. (Part 2 of 3.)

```
      // sort array, passing delegate for SortDescending
      DelegateBubbleSort.SortArray( elementArray,
         new DelegateBubbleSort.Comparator(
            SortDescending ) );

      DisplayResults();
   }

   // display the sorted array in sortedTextBox
   private void DisplayResults()
   {
      sortedTextBox.Clear();

      foreach ( int element in elementArray )
         sortedTextBox.Text += element + "\r\n";
   }

   // main entry point for application
   public static void Main( string[] args )
   {
      Application.Run( new BubbleSortForm() );
   }
}
```


BubbleSortForm
Original Order
Sorted Order
Create Data
Sort Ascending
Sort Descending

```
22
66
25
35
74
34
24
65
41
5
```

```
5
22
24
25
34
35
41
65
66
74
```

```
74
66
65
41
35
34
25
24
22
5
```

Fig. 10.25 Bubble-sort **Form** application. (Part 3 of 3.)

Methods **`ascendingButton_Click`** (lines 48–57) and **`descendingButton_Click`** (lines 66–75) are invoked when the user clicks the **Sort Ascending** and **Sort Descending** buttons, respectively. Method **`ascendingButton_Click`**, passes to **`DelegateBubbleSort`** method **`SortArray`** the unsorted **`elementArray`** (line 52) and a reference to method **`SortAscending`**. The syntax on lines 53–54

```
new DelegateBubbleSort.Comparator( SortAscending )
```

creates a **`Comparator`** delegate that contains a reference to method **`SortAscending`**. In method **`descendingButton_Click`**, lines 70–72 pass to method **`SortArray`** the unsorted array **`elementArray`** and a delegate reference to method **`SortDescending`**. We continue to use delegates in Chapters 12–14, when we discuss event handling and multithreading.

10.11 Operator Overloading

Manipulations on class objects are accomplished by sending messages (in the form of method calls) to the objects. This method-call notation is cumbersome for certain kinds of classes, especially mathematical classes. For these classes, it would be convenient to use C#'s rich set of built-in operators to specify object manipulations. In this section, we show how to enable C#'s operators to work with class objects—via a process called *operator overloading*.

Software Engineering Observation 10.13

Use operator overloading when it makes a program clearer than accomplishing the same operations with explicit method calls.

Software Engineering Observation 10.14

Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

C# enables the programmer to overload most operators to make them sensitive to the context in which they are used. Some operators are overloaded frequently, especially the assignment operator and various arithmetic operators, such as **`+`** and **`-`**. The job performed by overloaded operators also can be performed by explicit method calls, but operator notation often is more natural. Figure 10.27 provides an example of using operator overloading with a complex number class.

Class **`ComplexNumber`** (Fig. 10.26) overloads the plus (**`+`**), minus (**`-`**) and multiplication (**`*`**) operators to enable programs to add, subtract and multiply instances of class **`ComplexNumber`** using common mathematical notation. Lines 7–8 declare data members for the real and imaginary parts of the complex number.

```
1   // Fig. 10.26: ComplexNumber.cs
2   // Class that overloads operators for adding, subtracting
3   // and multiplying complex numbers.
4
5   public class ComplexNumber
6   {
```

Fig. 10.26 Overloading operators for complex numbers. (Part 1 of 3.)

```
   private int real;
   private int imaginary;

   // default constructor
   public ComplexNumber() {}

   // constructor
   public ComplexNumber( int a, int b )
   {
      Real = a;
      Imaginary = b;
   }

   // return string representation of ComplexNumber
   public override string ToString()
   {
      return "( " + real +
         ( imaginary < 0 ? " - " + ( imaginary * -1 ) :
         " + "  + imaginary ) + "i )";
   }

   // property Real
   public int Real
   {
      get
      {
         return real;
      }

      set
      {
         real = value;
      }

   } // end property Real

   // property Imaginary
   public int Imaginary
   {
      get
      {
         return imaginary;
      }

      set
      {
         imaginary = value;
      }

   } // end property Imaginary

```

Fig. 10.26 Overloading operators for complex numbers. (Part 2 of 3.)

```
58        // overload the addition operator
59        public static ComplexNumber operator + (
60           ComplexNumber x, ComplexNumber y )
61        {
62           return new ComplexNumber(
63              x.Real + y.Real, x.Imaginary + y.Imaginary );
64        }
65
66        // provide alternative to overloaded + operator
67        // for addition
68        public static ComplexNumber Add( ComplexNumber x,
69           ComplexNumber y )
70        {
71           return x + y;
72        }
73
74        // overload the subtraction operator
75        public static ComplexNumber operator - (
76           ComplexNumber x, ComplexNumber y )
77        {
78           return new ComplexNumber(
79              x.Real - y.Real, x.Imaginary - y.Imaginary );
80        }
81
82        // provide alternative to overloaded - operator
83        // for subtraction
84        public static ComplexNumber Subtract( ComplexNumber x,
85           ComplexNumber y )
86        {
87           return x - y;
88        }
89
90        // overload the multiplication operator
91        public static ComplexNumber operator * (
92           ComplexNumber x, ComplexNumber y )
93        {
94           return new ComplexNumber(
95              x.Real * y.Real - x.Imaginary * y.Imaginary,
96              x.Real * y.Imaginary + y.Real * x.Imaginary );
97        }
98
99        // provide alternative to overloaded * operator
100       // for multiplication
101       public static ComplexNumber Multiply( ComplexNumber x,
102          ComplexNumber y )
103       {
104          return x * y;
105       }
106
107    } // end class ComplexNumber
```

Fig. 10.26 Overloading operators for complex numbers. (Part 3 of 3.)

Lines 59–64 overload the plus operator (**+**) to perform addition of **ComplexNumber**s. Keyword ***operator*** followed by an operator indicates that a method overloads the specified operator. Methods that overload binary operators must take two arguments. The first argument is the left operand, and the second argument is the right operand. Class **`ComplexNumber`**'s overloaded plus operator takes two **`ComplexNumber`** references as arguments and returns a **`ComplexNumber`** that represents the sum of the arguments. Note that this method is marked **`public`** and **`static`**, which is required for overloaded operators. The body of the method (lines 62–63) performs the addition and returns the result as a new **`ComplexNumber`** reference.

Software Engineering Observation 10.15

Overload operators to perform the same function or similar functions on class objects as the operators perform on objects of built-in types. Avoid non-intuitive use of operators.

Software Engineering Observation 10.16

At least one argument of an operator overload method must be a reference to an object of the class in which the operator is overloaded. This prevents programmers from changing how operators work on built-in types.

Not all .NET languages support operator overloading. Therefore, to ensure that our **`ComplexNumber`** class can be used in other .NET languages, we must provide an alternative method for performing addition of **`ComplexNumber`**s. Method **`Add`** (lines 68–72) provides this alternative means to add **`ComplexNumber`**s. Lines 75–105 provide overloaded operators and alternative methods for subtracting and multiplying **`ComplexNumber`**s.

Class **`ComplexTest`** (Fig. 10.27) provides a user interface for adding, subtracting and multiplying **`ComplexNumber`**s. Method **`firstButton_Click`** and method **`secondButton_Click`** each read a **`ComplexNumber`** from textboxes **`realTextBox`** and **`imaginaryTextBox`**. Method **`addButton_Click`** (lines 56–59), method **`subtractButton_Click`** (lines 62–66) and method **`multiplyButton_Click`** (lines 69–73) use overloaded operators of class **`ComplexNumber`** to perform addition, subtraction and multiplication.

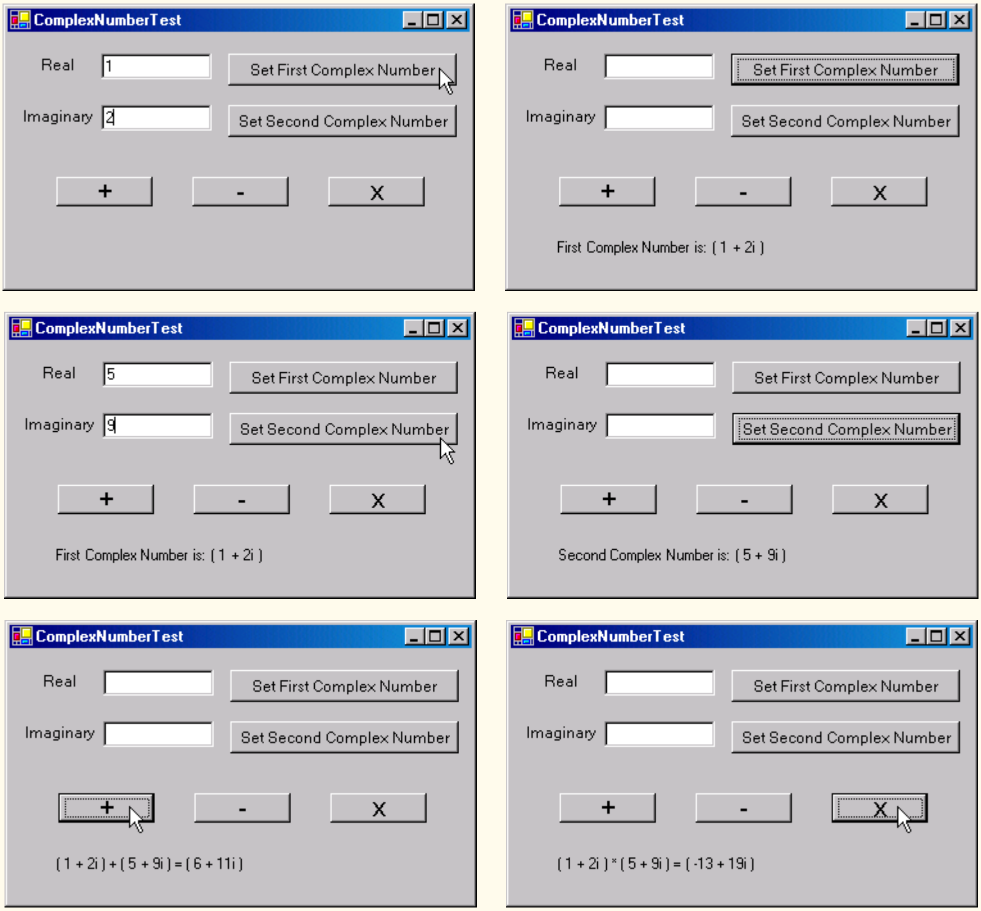
```
 1   // Fig 10.27: OperatorOverloading.cs
 2   // An example that uses operator overloading
 3
 4   using System;
 5   using System.Drawing;
 6   using System.Collections;
 7   using System.ComponentModel;
 8   using System.Windows.Forms;
 9   using System.Data;
10
11   public class ComplexTest : System.Windows.Forms.Form
12   {
13      private System.Windows.Forms.Label realLabel;
14      private System.Windows.Forms.Label imaginaryLabel;
15      private System.Windows.Forms.Label statusLabel;
16
17      private System.Windows.Forms.TextBox realTextBox;
18      private System.Windows.Forms.TextBox imaginaryTextBox;
19
```

Fig. 10.27 Using operator overloading. (Part 1 of 3.)

```
      private System.Windows.Forms.Button firstButton;
      private System.Windows.Forms.Button secondButton;
      private System.Windows.Forms.Button addButton;
      private System.Windows.Forms.Button subtractButton;
      private System.Windows.Forms.Button multiplyButton;

      private ComplexNumber x = new ComplexNumber();
      private ComplexNumber y = new ComplexNumber();

      [STAThread]
      static void Main()
      {
         Application.Run( new ComplexTest() );
      }

      private void firstButton_Click(
         object sender, System.EventArgs e )
      {
         x.Real = Int32.Parse( realTextBox.Text );
         x.Imaginary = Int32.Parse( imaginaryTextBox.Text );
         realTextBox.Clear();
         imaginaryTextBox.Clear();
         statusLabel.Text = "First Complex Number is: " + x;
      }

      private void secondButton_Click(
         object sender, System.EventArgs e )
      {
         y.Real = Int32.Parse( realTextBox.Text );
         y.Imaginary = Int32.Parse( imaginaryTextBox.Text );
         realTextBox.Clear();
         imaginaryTextBox.Clear();
         statusLabel.Text = "Second Complex Number is: " + y;
      }

      // add complex numbers
      private void addButton_Click( object sender, System.EventArgs e )
      {
         statusLabel.Text = x + " + " + y + " = " + ( x + y );
      }

      // subtract complex numbers
      private void subtractButton_Click(
         object sender, System.EventArgs e )
      {
         statusLabel.Text = x + " - " + y + " = " + ( x - y );
      }

      // multiply complex numbers
      private void multiplyButton_Click(
         object sender, System.EventArgs e )
      {
```

Fig. 10.27 Using operator overloading. (Part 2 of 3.)

```
         statusLabel.Text = x + " * " + y + " = " + ( x * y );
      }

   } // end class ComplexTest
```



Real 1
Imaginary 2

First Complex Number is: (1 + 2i)

Real 5
Imaginary 9

First Complex Number is: (1 + 2i)

Second Complex Number is: (5 + 9i)

(1 + 2i) + (5 + 9i) = (6 + 11i)

(1 + 2i) * (5 + 9i) = (-13 + 19i)

Fig. 10.27 Using operator overloading. (Part 3 of 3.)

SUMMARY

- Polymorphism enables us to write programs in a general fashion to handle a wide variety of existing and future related classes.
- One means of processing objects of many different types is to use a **`switch`** structure to perform an appropriate action on each object based on that object's type.
- Polymorphic programming can eliminate the need for **`switch`** logic.
- When we override a base class's method in a derived class, we hide the base class's implementation of that method.
- With polymorphism, new types of objects not even envisioned when a system is created may be added without modification to the system (other than the new class itself).

- Polymorphism allows one method call to perform different actions, depending on the type of the object receiving the call. The same message assumes “many forms”—hence, the term polymorphism.
- With polymorphism, the programmer can deal in generalities and let the executing program concern itself with the specifics.
- Any class with an abstract method in it must, itself, be declared `abstract`.
- A program cannot instantiate objects of `abstract` classes, but can declare references to `abstract` classes. Such references can manipulate polymorphically instances of the derived classes.
- A method that is not declared `virtual` cannot be overridden in a derived class.
- Methods that are declared `static` and or `private` are implicitly non-virtual.
- A `sealed` class cannot be a base class (i.e., a class cannot inherit from a `sealed` class).
- In C#, it is impossible to pass a method reference directly as an argument to another method. To address this problem, C# allows the creation of delegates, which are classes that encapsulate a set of references to methods.
- C# enables the programmer to overload most operators to make them sensitive to the context in which they are used.
- Methods that overload binary operators must take two arguments. The first argument is the left operand, and the second argument is the right operand.

TERMINOLOGY

`abstract` base class
abstract class
abstract method
`abstract` method
cast
class declared `sealed`
class hierarchy
concrete class
delegate
information hiding
inheritance
inheritance hierarchy
interface
`InvalidCastException`
“is-a” relationship
method reference
multicast delegate
object-oriented programming (OOP)
operator overloading
`override` keyword
polymorphic programming
polymorphism
reference type
references to abstract base class
`sealed` class
singlecast delegate
`switch` logic
`virtual` method

SELF-REVIEW EXERCISES

10.1 Fill in the blanks in each of the following statements:

a) Treating a base-class object as a __________ can cause errors.
b) Polymorphism helps eliminate unnecessary __________ logic.
c) If a class contains one or more `abstract` methods, it is an __________ class.
d) Classes from which objects can be instantiated are called __________ classes.
e) Classes declared with keyword __________ cannot be inherited.
f) An attempt to cast an object to one of its derived types can cause an __________.
g) Polymorphism involves using a base-class reference to manipulate __________.
h) Abstract classes are declared with the __________ keyword.
i) Class members can be overridden only with the __________ keyword.
j) __________ are classes that encapsulate references to methods.

10.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) All methods in an abstract base class must be declared **`abstract`**.
b) Referring to a derived-class object with a base-class reference is dangerous.
c) A class with an abstract method must be declared **`abstract`**.
d) Methods that are declared **`abstract`** still must be implemented when they are declared.
e) Classes declared with the **`sealed`** keyword cannot be base classes.
f) Polymorphism allows programmers to manipulate derived classes with references to base classes.
g) Polymorphic programming can eliminate the need for unnecessary **`switch`** logic.
h) Use keyword **`abstract`** to declare an abstract method.
i) The delegate's declaration must specify its implementation.

ANSWERS TO SELF-REVIEW EXERCISES

10.1 a) derived-class object. b) **`switch`**. c) abstract. d) concrete. e) **`sealed`**. f) **`InvalidCastException`**. g) derived-class objects. h) **`abstract`**. i) **`override`**. j) Delegates

10.2 a) False. Not all methods in an abstract class must be declared **`abstract`**. b) False. Referring to a base-class object with a derived-class reference is dangerous. c) True. d) False. Methods that are declared **`abstract`** do not need to be implemented, except in the derived, concrete class. e) True. f) True. g) True. h) False. Use keyword **`abstract`** to declare an abstract class. i) False. The delegate's declaration specifies only a method signature (method name, parameters and return value).

EXERCISES

10.3 How is it that polymorphism enables you to program "in the general" rather than "in the specific?" Discuss the key advantages of programming "in the general."

10.4 Discuss the problems of programming with **`switch`** logic. Explain why polymorphism can be an effective alternative to using **`switch`** logic.

10.5 Distinguish between inheriting services and inheriting implementation. How do inheritance hierarchies designed for inheriting services differ from those designed for inheriting implementation?

10.6 Modify the payroll system of Fig. 10.10–Fig. 10.14 to add **`Private`** instance variables **`birthDate`** (use class **`Day`** from Fig 8.8) and **`departmentCode`** (an **`int`**) to class **`Employee`**. Assume this payroll is processed once per month. Create an array of **`Employee`** references to store the various employee objects. In a loop, calculate the payroll for each **`Employee`** (polymorphically) and add a $100.00 bonus to the person's payroll amount if this is the month in which the **`Employee`**'s birthday occurs.

10.7 Implement the **`Shape`** hierarchy shown in Fig. 9.3. Each **`TwoDimensionalShape`** should contain method **`Area`** to calculate the area of the two-dimensional shape. Each **`ThreeDimensionalShape`** should have methods **`Area`** and **`Volume`** to calculate the surface area and volume of the three-dimensional shape, respectively. Create a program that uses an array of **`Shape`** references to objects of each concrete class in the hierarchy. The program should output the **`string`** representation of each object in the array. Also, in the loop that processes all the shapes in the array, determine whether each shape is a **`TwoDimensionalShape`** or a **`ThreeDimensionalShape`**. If a shape is a **`TwoDimensionalShape`**, display its **`Area`**. If a shape is a **`ThreeDimensionalShape`**, display its **`Area`** and **`Volume`**.

10.8 Reimplement the program of Exercise 10.7 such that classes **`TwoDimensionalShape`** and **`ThreeDimensionalShape`** implement an **`IShape`** interface, rather than extending **`abstract`** class **`Shape`**.

11

Exception Handling

Objectives

- To understand exceptions and error handling.
- To use **`try`** blocks to delimit code in which exceptions may occur.
- To **`throw`** exceptions.
- To use **`catch`** blocks to specify exception handlers.
- To use the **`finally`** block to release resources.
- To understand the C# exception-class hierarchy.
- To create programmer-defined exceptions.

It is common sense to take a method and try it. If it fails,
admit it frankly and try another. But above all, try something.
Franklin Delano Roosevelt

O! throw away the worser part of it,
And live the purer with the other half.
William Shakespeare

If they're running and they don't look where they're going
I have to come out from somewhere and catch them.
Jerome David Salinger

And oftentimes excusing of a fault
Doth make the fault the worse by the excuse.
William Shakespeare

I never forget a face, but in your case I'll make an exception.
Groucho (Julius Henry) Marx

Outline

11.1 Introduction
11.2 Exception Handling Overview
11.3 Example: `DivideByZeroException`
11.4 .NET `Exception` Hierarchy
11.5 `finally` Block
11.6 `Exception` Properties
11.7 Programmer-Defined Exception Classes
11.8 Handling Overflows with Operators `checked` and `unchecked`

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

11.1 Introduction

In this chapter, we introduce *exception handling*. An *exception* is an indication of a problem that occurs during a program's execution. The name "exception" comes from the fact that although a problem can occur, the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the "exception to the rule" is that a problem occurs. Exception handling enables programmers to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continue executing as if no problem was encountered. A more severe problem may prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate in a controlled manner. The features presented in this chapter enable programmers to write clear, robust and more *fault-tolerant programs*.

The style and details of exception handling in C# are based in part on the work of Andrew Koenig and Bjarne Stroustrup, as presented in their paper, "Exception Handling for C++ (revised)."[1] C#'s designers implemented an exception-handling mechanism similar to that used in C++, with Koenig's and Stroustrup's work as a model.

This chapter begins with an overview of exception-handling concepts, then demonstrates basic exception-handling techniques. The chapter continues with an overview of the exception-handling class hierarchy. Programs typically request and release resources (such as files on disk) during program execution. Often, these resources are in limited supply or can be used by only one program at a time. We demonstrate a part of the exception-handling mechanism that enables a program to use a resource, then guarantees that the program releases the resource for use by other programs. The chapter continues with an example that demonstrates several properties of class **`System.Exception`** (the base class of all exception classes), followed by an example that shows programmers how to create and use their own exception classes. The chapter concludes with a practical application of exception handling in which a program handles exceptions generated by arithmetic calculations that result in out-of-range values for a particular data type—a condition known as *arithmetic overflow*.

---

[1] Koenig, A. and B. Stroustrup "Exception Handling for C++ (revised)", *Proceedings of the Usenix C++ Conference*, 149-176, San Francisco, April 1990.

11.2 Exception Handling Overview

The logic of the program frequently tests conditions that determine how program execution proceeds. Consider the following pseudocode:

```
Perform a task

If the preceding task did not execute correctly
    Perform error processing

Perform next task

If the preceding task did not execute correctly
    Perform error processing
…
```

In this pseudocode, we begin by performing a task. We then test whether that task executed correctly. If not, we perform error processing. Otherwise we start the entire process again and continue with the next task. Although this form of error handling logic works, intermixing the logic of the program with the error-handling logic can make the program difficult to read, modify, maintain and debug—especially in large applications. In fact, if many of the potential problems occur infrequently, intermixing program logic and error handling can degrade the performance of the program, because the program must test extra conditions to determine whether the next task can be performed.

Exception handling enables the programmer to remove error-handling code from the "main line" of the program's execution. This improves program clarity and enhances modifiability. Programmers can decide to handle whatever exceptions they choose—all types of exceptions, all exceptions of a certain type or all exceptions of a group of related types. Such flexibility reduces the likelihood that errors will be overlooked and thereby increases a program's robustness.

Testing and Debugging Tip 11.1

Exception handling helps improve a program's fault tolerance. When it is easy to write error-processing code, programmers are more likely to use it.

Software Engineering Observation 11.1

Although it is possible to do so, do not use exception for conventional flow of control. It is difficult to keep track of a larger number of exception cases and programs with a large number of exception cases are hard to read and maintain.

Exception handling is designed to process *synchronous errors*—errors that occur during the normal program flow of control. Common examples of these errors are out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid method parameters and running out of available memory. Exception handling is not designed to process *asynchronous* events, such as disk I/O completions, network message arrivals, mouse clicks, keystrokes and the like.

Good Programming Practice 11.1

Avoid using exception handling for purposes other than error handling, because this can reduce program clarity.

With programming languages that do not support exception handling, programmers often delay the writing of error-processing code and sometimes simply forget to include it. This results in less robust software products. C# enables the programmer to deal with exception handling easily from the inception of a project. Still, the programmer must put considerable effort into incorporating an exception-handling strategy into software projects.

Software Engineering Observation 11.2

Try to incorporate the exception-handling strategy into a system from the inception of the design process. Adding effective exception handling after a system has been implemented can be difficult.

Software Engineering Observation 11.3

In the past, programmers used many techniques to implement error-processing code. Exception handling provides a single, uniform technique for processing errors. This helps programmers working on large projects to understand each other's error-processing code.

The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements, such as methods, constructors, assemblies and classes. Rather than internally handling problems that occur, such software elements often use exceptions to notify programs when problems occur. This enables programmers to implement customized error handling for each application.

Common Programming Error 11.1

Aborting a program component could leave a resource—such as file stream or I/O device—in a state in which other programs are unable to acquire the resource. This is known as a "resource leak."

Performance Tip 11.1

When no exceptions occur, exception-handling code incurs little or no performance penalties. Thus, programs that implement exception handling operate more efficiently than programs that perform error handling throughout the program logic.

Performance Tip 11.2

Exception handling should be used only for problems that occur infrequently. As a "rule of thumb," if a problem occurs at least 30% of the time when a particular statement executes, the program should test for the error inline; otherwise, the overhead of exception handling will cause the program to execute more slowly.[2]

Software Engineering Observation 11.4

Methods with common error conditions should return **`null`** *(or another appropriate value) rather than throwing exceptions. A program calling such a method simply can check the return value to determine success or failure of the method call.*[3]

Complex applications normally consist of predefined software components (such as those defined in the .NET Framework) and components specific to the application that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application will process a problem that occurs. Exception handling simplifies combining software components and having them work together effectively by enabling predefined components to communicate problems that occur to application-specific components, which can then process the problems in an application-specific manner.

---

[2] "Best Practices for Handling Exceptions [C#]," *.NET Framework Developer's Guide*, Visual Studio .NET Online Help.

[3] "Best Practices for Handling Exceptions [C#]."

Exception handling is geared to situations in which the method that detects an error is unable to handle it. Such a method *throws an exception*. There is no guarantee that there will be an *exception handler*—code that executes when the program detects an exception—to process that kind of exception. If there is, the exception will be *caught* and *handled*. The result of an *uncaught exception* depends on whether the program executes in debug mode or standard execution mode. In debug mode, when the program detects an uncaught exception, a dialog box appears that enables the programmer to view the problem in the debugger or continue program execution by ignoring the problem that occurred. In standard execution mode, a Windows application presents a dialog that enables the user to continue or terminate program execution, and a console application presents a dialog that enables the user to open the program in the debugger or terminate program execution.

C# uses *`try` blocks* to enable exception handling. A `try` block consists of keyword `try` followed by braces (`{}`) that define a block of code in which exceptions may occur. The `try` block encloses statements that could cause exceptions. Immediately following the `try` block are zero or more *`catch` blocks* (also called *`catch` handlers*). Each `catch` handler specifies in parentheses an exception parameter that represents the type of exception the `catch` handler can handle. If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with a caught exception object. Optionally, programmers can include a *parameterless `catch` handler* that catches all exception types. After the last `catch` handler, an optional *`finally` block* contains code that always executes, regardless of whether an exception occurs.

Common Programming Error 11.2

The parameterless `catch` handler must be the last `catch` handler following a particular `try` block; otherwise a syntax error occurs.

When a method called in a program detects an exception or when the Common Language Runtime detects a problem, the method or CLR *throws an exception*. The point in the program at which an exception occurs is called the *throw point*—an important location for debugging purposes (as we demonstrate in Section 11.6). Exceptions are objects of classes that extend class *`Exception`* of namespace `System`. If an exception occurs in a `try` block, the `try` block *expires* (i.e., terminates immediately) and program control transfers to the first `catch` handler (if there is one) following the `try` block. C# is said to use the *termination model of exception handling*, because the `try` block enclosing a thrown exception expires immediately when that exception occurs.[4] As with any other block of code, when a `try` block terminates, local variables defined in the block go out of scope. Next, the CLR searches for the first `catch` handler that can process the type of exception that occurred. The CLR locates the matching `catch` by comparing the thrown exception's type to each `catch`'s exception-parameter type until the CLR finds a match. A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type. When a **`catch`** handler finishes processing, local variables defined within the **`catch`** handler (including the **`catch`** parameter) go out of scope. If a match occurs, code contained within the matching **`catch`** handler is executed. All remaining **`catch`** handlers that correspond to the **`try`** block are ignored and execution resumes at the first line of code after the **`try`**/**`catch`** sequence.

---

[4] Some languages use the *resumption model of exception handling*, in which, after the handling of the exception, control returns to the point at which the exception was thrown and execution resumes from that point.

If no exceptions occur in a **`try`** block, the CLR ignores the exception handlers for that block. Program execution resumes with the next statement after the **`try`**/**`catch`** sequence. If an exception that occurs in a **`try`** block has no matching **`catch`** handler, or if an exception occurs in a statement that is not in a **`try`** block, the method containing that statement terminates immediately and the CLR attempts to locate an enclosing **`try`** block in a calling method. This process is called *stack unwinding* (discussed in Section 11.6).

11.3 Example: `DivideByZeroException`

Let us consider a simple example of exception handling. The application in Fig. 11.1 uses **`try`** and **`catch`** to specify a block of code that may throw exceptions and to handle those exceptions if they occur. The application displays two **`TextBox`**es in which the user can type integers. When the user presses the **Click To Divide** button, the program invokes method **`divideButton_Click`** (lines 46–84), which obtains the user's input, converts the input values to type **`int`** and divides the first number (**`numerator`**) by the second number (**`denominator`**). Assuming that the user provides integers as input and does not specify 0 as the denominator for the division, **`divideButton_Click`** displays the division result in **`outputLabel`**. However, if the user inputs a non-integer value or supplies 0 as the denominator, an exception occurs. This program demonstrates how to catch these exceptions.

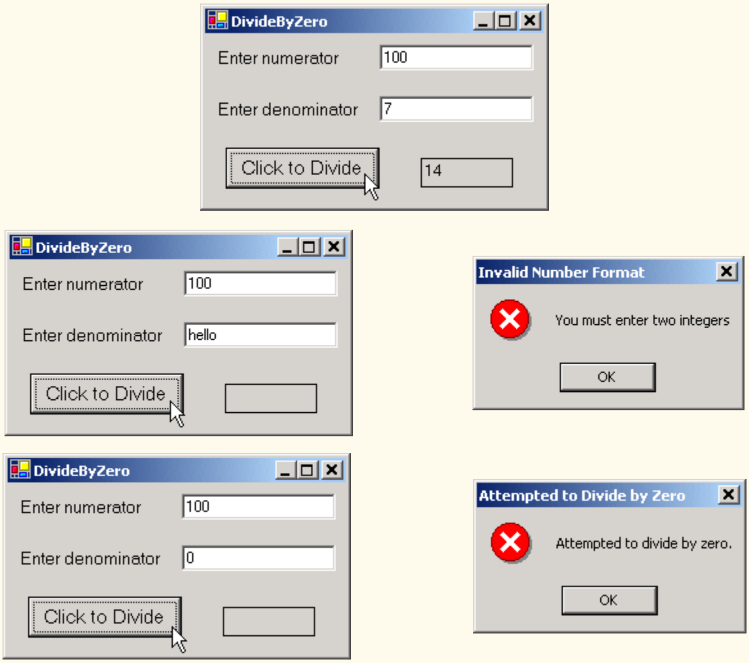
```
 1  // Fig 11.1: DivideByZeroTest.cs
 2  // Basics of C# exception handling.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  // class demonstrates how to handle exceptions from
12  // division by zero in integer arithmetic and from
13  // improper numeric formatting
14  public class DivideByZeroTest : System.Windows.Forms.Form
15  {
16     private System.Windows.Forms.Label numeratorLabel;
17     private System.Windows.Forms.TextBox numeratorTextBox;
18
19     private System.Windows.Forms.Label denominatorLabel;
20     private System.Windows.Forms.TextBox denominatorTextBox;
21
22     private System.Windows.Forms.Button divideButton;
```

Fig. 11.1 Exception handlers for **`FormatException`** and **`DivideByZeroException`**. (Part 1 of 3.)

```
   private System.Windows.Forms.Label outputLabel;

   // required designer variable
   private System.ComponentModel.Container components = null;

   // default constructor
   public DivideByZeroTest()
   {
      // required for Windows Form Designer support
      InitializeComponent();
   }

   // main entry point for the application
   [STAThread]
   static void Main()
   {
      Application.Run( new DivideByZeroTest() );
   }

   // Visual Studio .NET generated code

   // obtain integers input by user and divide numerator
   // by denominator
   private void divideButton_Click(
      object sender, System.EventArgs e )
   {
      outputLabel.Text = "";

      // retrieve user input and call Quotient
      try
      {
         // Convert.ToInt32 generates FormatException if
         // argument is not an integer
         int numerator = Convert.ToInt32( numeratorTextBox.Text );
         int denominator =
            Convert.ToInt32( denominatorTextBox.Text );

         // division generates DivideByZeroException if
         // denominator is 0
         int result = numerator / denominator;

         outputLabel.Text = result.ToString();

      } // end try

      // process invalid number format
      catch ( FormatException )
      {
         MessageBox.Show( "You must enter two integers",
            "Invalid Number Format",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      }
```

Fig. 11.1 Exception handlers for **`FormatException`** and **`DivideByZeroException`**. (Part 2 of 3.)

```

      // user attempted to divide by zero
      catch ( DivideByZeroException divideByZeroException )
      {
         MessageBox.Show( divideByZeroException.Message,
            "Attempted to Divide by Zero",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      }

   } // end method divideButton_Click

} // end class DivideByZeroTest
```



DivideByZero
Enter numerator 100
Enter denominator 7
14

DivideByZero
Enter numerator 100
Enter denominator hello

Invalid Number Format
You must enter two integers

DivideByZero
Enter numerator 100
Enter denominator 0

Attempted to Divide by Zero
Attempted to divide by zero.

Fig. 11.1 Exception handlers for `FormatException` and `DivideByZeroException`. (Part 3 of 3.)

Before we discuss the program details, consider the sample output windows in Fig. 11.1. The first window shows a successful calculation in which the user inputs the numerator `100` and the denominator `7`. Note that the result (`14`) is an integer, because integer division always yields integer results. The next two windows show the result of inputting a non-integer value—in this case, the user input **`"hello"`** in the second **`TextBox`**. When the user presses **Click To Divide**, the program attempts to convert the **`string`**s the user input into **`int`** values with method **`Convert.ToInt32`**. If the argument to **`Convert.ToInt32`** is not a valid representation of an integer (in this case a valid string representation of an integer), the method generates a ***`FormatException`*** (namespace **`System`**). The program detects the exception and displays an error message dialog, indicating that the user must enter two integers. The last two output windows demonstrate the result after an attempt to divide by zero. In integer arithmetic, the CLR automatically tests for division by zero and generates a ***`DivideByZeroException`*** (namespace **`System`**) if the denominator is zero. The program detects the exception and displays an error-message dialog, indicating an attempt to divide by zero.[5]

Let us consider the user interactions and flow of control that yield the results shown in the sample output windows. The user inputs values into the **`TextBox`**es that represent the numerator and denominator, then presses **ClicktoDivide**. At this point, the program invokes method **`divideButton_Click`** (lines 46–84). Line 49 assigns the empty **`string`** to **`outputLabel`** to clear any prior result, because the program is about to attempt a new calculation. Lines 52–66 define a **`try`** block that encloses the code that can throw exceptions, as well as the code that should not execute if an exception occurs. For example, the program should not display a new result in **`outputLabel`** (line 64) unless the calculation (line 62) completes successfully. Remember that the **`try`** block terminates immediately if an exception occurs, so the remaining code in the **`try`** block will not execute.

The two statements that read the integers from the **`TextBox`**es (lines 56–58) each call method **`Convert.ToInt32`** to convert **`string`**s to **`int`** values. This method throws a **`FormatException`** if it cannot convert its **`string`** argument to an integer. If lines 56–58 properly convert the values (i.e., no exceptions occur), then line 62 divides the **`numerator`** by the **`denominator`** and assigns the result to variable **`result`**. If the denominator is zero, line 62 causes the CLR to throw a **`DivideByZeroException`**. If line 62 does not cause an exception, then line 64 displays the result of the division. If no exceptions occur in the **`try`** block, the program successfully completes the **`try`** block by reaching line 66 by ignoring the **`catch`** handlers at lines 69–74 and 77–82—the program execution continues with the first statement following the **`try`**/**`catch`** sequence. In this example, the program reaches the end of event handler **`divideButton_Click`**, so the method terminates, and the program awaits the next user interaction.

Immediately following the **`try`** block are two **`catch`** handlers (also called **`catch`** handlers)—lines 69–74 define the exception handler for a **`FormatException`** and lines 77–82 define the exception handler for the **`DivideByZeroException`**. Each **`catch`** handler begins with keyword **`catch`** followed by an exception parameter in parenthesis that specifies the type of exception handled by the **`catch`** handler. The exception-handling code appears in the **`catch`** handler. In general, when an exception occurs in a **`try`** block, a **`catch`** handler catches the exception and handles it. In Fig. 11.1, the first **`catch`** handler specifies that it catches **`FormatException`**s (thrown by method **`Convert.ToInt32`**) and the second **`catch`** handler specifies that it catches **`DivideByZeroException`**s (thrown by the CLR). Only the matching **`catch`** handler executes if an exception occurs. Both the exception handlers in this example display an error-message dialog. When program control reaches the end of a **`catch`** handler, the program considers the exception as having been handled, and program control continues with the first statement after the `try`/`catch` sequence (the end of the method in this example).

---

[5] The Common Language Runtime allows floating-point division by zero, which produces a positive or negative infinity result, depending on whether the numerator is positive or negative. Dividing zero by zero is a special case that results in a value called “not a number.” Programs can test for these results using constants for positive infinity (***`PositiveInfinity`***), negative infinity (***`NegativeInfinity`***) and not a number (***`NaN`***) that are defined in structures **`Double`** (for **`double`** calculations) and **`Single`** (for **`float`** calculations).

In the second sample output, the user input `hello` as the denominator. When lines 57–58 execute, `Convert.ToInt32` cannot convert this `string` to an `int`, so `Convert.ToInt32` creates a `FormatException` object and throws it to indicate that the method was unable to convert the `string` to an `int`. When an exception occurs, the `try` block expires (terminates). Any local variables defined in the `try` block go out of scope; therefore, those variables are not available to the exception handlers. Next, the CLR attempts to locate a matching `catch` handler, starting with the `catch` at line 69. The program compares the type of the thrown exception (`FormatException`) with the type in parentheses following keyword `catch` (also `FormatException`). A match occurs, so that exception handler executes and the program ignores all other exception handlers following the corresponding `try` block. Once the `catch` handler finishes processing, local variables defined within the `catch` handler go out of scope. If a match did not occur, the program compares the type of the thrown exception with the next `catch` handler in sequence and repeats the process until a match is found.

Software Engineering Observation 11.5

Enclose in a `try` block a significant logical section of the program in which several statements can throw exceptions, rather than using a separate `try` block for every statement that throws an exception. However, for proper exception-handling granularity, each `try` block should enclose a section of code small enough, that when an exception occurs, the specific context is known and the `catch` handlers can process the exception properly.

Common Programming Error 11.3

Attempting to access a `try` block's local variables in one of that `try` block's associated `catch` handlers is a syntax error. Before a corresponding `catch` handler can execute, the `try` block expires, and its local variables go out of scope.

Common Programming Error 11.4

Specifying a comma-separated list of exception parameters in a `catch` handler is a syntax error. Each `catch` can have only one exception parameter.

In the third sample output, the user input `0` as the denominator. When line 62 executes, the CLR throws a `DivideByZeroException` object to indicate an attempt to divide by zero. Once again, the `try` block terminates immediately upon encountering the exception, and the program attempts to locate a matching `catch` handler, starting from the `catch` handler at line 69. The program compares the type of the thrown exception (`DivideByZeroException`) with the type in parentheses following keyword `catch` (`FormatException`). In this case, there is no match, because they are not the same exception types and because `FormatException` is not a base class of `DivideByZeroException`. So, the program proceeds to line 77 and compares the type of the thrown exception (`DivideByZeroException`) with the type in parentheses following keyword `catch` (`DivideByZeroException`). A match occurs, so that exception handler executes. Line 79 in this handler uses property ***`Message`*** of class `Exception` to display the error message to the user. If there were additional `catch` handlers, the program would ignore them.

11.4 .NET `Exception` Hierarchy

The exception-handling mechanism allows only objects of class **`Exception`** and its derived classes to be thrown and caught[6]. This section overviews several of the .NET Framework's exception classes. In addition, we discuss how to determine whether a particular method throws exceptions.

Class **`Exception`** of namespace **`System`** is the base class of the .NET Framework exception hierarchy. Two of the most important derived classes of **`Exception`** are ***`ApplicationException`*** and ***`SystemException`***. **`ApplicationException`** is a base class programmers can extend to create exception data types that are specific to their applications. We discuss creating programmer-defined exception classes in Section 11.7. Programs can recover from most **`ApplicationException`**s and continue execution.

The CLR can generate **`SystemException`**s at any point during the execution of the program. Many of these exceptions can be avoided by coding properly. These are called *runtime exceptions* and they derive from class ***`SystemException`***. For example, if a program attempts to access an out-of-range array subscript, the CLR throws an exception of type **`IndexOutOfRangeException`** (a class derived from **`SystemException`**). Similarly, a runtime exception occurs when a program uses an object reference to manipulate an object that does not yet exist (i.e., the reference has a **`null`** value). Attempting to use such a **`null`** reference causes a ***`NullReferenceException`*** (another type of **`SystemException`**). According to Microsoft's "Best Practices for Handling Exceptions [C#],"[7] programs typically cannot recover from most exceptions the CLR throws. Therefore, programs generally should not throw or catch **`SystemException`**s. [*Note:* For a complete list of derived classes of **`Exception`**, look up "**`Exception`** class" in the **Index** of the Visual Studio .NET online documentation.]

A benefit of using the exception-class hierarchy is that a **`catch`** handler can catch exceptions of a particular type or can use a base-class type to catch exceptions in a hierarchy of related exception types. For example, a **`catch`** handler that specifies an exception parameter of type **`Exception`** also can catch exceptions of all classes that extend **`Exception`**, because **`Exception`** is the base class of all exception classes. This allows for polymorphic processing of related exceptions. The benefit of the latter approach is that the exception handler can use the exception parameter to manipulate the caught exception. If the exception handler does not need access to the caught exception, the exception parameter may be omitted. If no exception type is specified, the catch handler will catch all exceptions.

Using inheritance with exceptions enables an exception handler to catch related exceptions with a concise notation. An exception handler certainly could catch each derived-class exception type individually, but catching the base-class exception type is more concise. However, this makes sense only if the handling behavior is the same for a base class and derived classes. otherwise, catch each derived-class exception individually.

---

[6] Actually, it is possible to **`catch`** exceptions of types that are not derived from class **`Exception`** using the parameterless **`catch`** handler. This is useful for handling exceptions from code written in other languages that do not require all exception types to derive from class **`Exception`** in the .NET framework.

[7] "Best Practices for Handling Exceptions [C#]," *.NET Framework Developer's Guide*, Visual Studio .NET Online Help.

At this point, we know that there are many different exception types. We also know that methods and the CLR can both throw exceptions. But, how do we determine that an exception could occur in a program? For methods in the .NET Framework classes, we can look at the detailed description of the methods in the online documentation. If a method throws an exception, its description contains a section called "Exceptions" that specifies the types of exceptions thrown by the method and briefly describes potential causes for the exceptions. For example, look up "`Convert.ToInt32` method" in the index of the Visual Studio .NET online documentation. In the document that describes the method, click the link "`public static int ToInt32(string);`." In the document that appears, the "Exceptions" section indicates that method `Convert.ToInt32` throws three exception types—`ArgumentException`, `FormatException` and `OverflowException`—and describes the reason that each exception type occurs.

Software Engineering Observation 11.6

If a method is capable of throwing exceptions, statements that invoke that method should be placed in `try` *blocks and those exceptions should be caught and handled.*

Determining when the CLR throws exceptions is more difficult. Typically, such information appears in the *C# Language Specification*, which is located in the online documentation. To access the language specification, select **Contents…** from the **Help** menu in Visual Studio. In the **Contents** window, expand **Visual Studio .NET**, **Visual Basic and Visual C#**, **Reference**, **Visual C# Language** and **C# Language Specification**.

The language specification defines the syntax of the language and specifies cases in which exceptions are thrown. For example, in Fig. 11.1, we demonstrated that the CLR throws a `DivideByZeroException` when a program attempts to divide by zero in integer arithmetic. The language specification, Section 7.7.2 discusses the division operator and its `Exception`s. In this section, you will find the details of when a `DivideByZeroException` occurs.

11.5 `finally` Block

Programs frequently request and release resources dynamically (i.e., at execution time). For example, a program that reads a file from disk first requests the opening of that file. If that request succeeds, the program reads the contents of the file. Operating systems typically prevent more than one program from manipulating a file at once. Therefore, when a program finishes processing a file, the program normally closes the file (i.e., releases the resource). This enables other programs to use the file. Closing the file helps prevent the *resource leak*, in which the file resource is unavailable to other programs because a program using the file never closed it. Programs that obtain certain types of resources (such as files) must return those resources explicitly to the system to avoid resource leaks.

In programming languages, like C and C++, in which the programmer is responsible for dynamic memory management, the most common type of resource leak is a *memory leak*. This happens when a program allocates memory (as we do with operator `new` in C#), but does not deallocate the memory when the memory is no longer needed in the program. In C#, this normally is not an issue, because the CLR performs "garbage collection" of memory no longer needed by an executing program. However, other kinds of resource leaks (such as the unclosed file mentioned previously) can occur in C#.

Testing and Debugging Tip 11.2

The CLR does not completely eliminate memory leaks. The CLR will not garbage-collect an object until the program has no more references to that object. Thus, memory leaks can occur if programmers erroneously keep references to unwanted objects.

Most resources that require explicit release have potential exceptions associated with the processing of the resource. For example, a program that processes a file might receive `IOException`s during the processing. For this reason, file-processing code normally appears in a `try` block. Regardless of whether a program successfully processes a file, the program should close the file when the file is no longer needed. Suppose a program places all resource-request and resource-release code in a `try` block. If no exceptions occur, the `try` block executes normally and releases the resources after using them. However, if an exception occurs, the `try` block may expire before the resource-release code can execute. We could duplicate all resource-release code in the `catch` handlers, but this makes the code more difficult to modify and maintain.

C#'s exception handling mechanism provides the ***`finally`*** block, which is guaranteed to execute if program control enters the corresponding `try` block. The `finally` block executes regardless of whether that `try` block executes successfully or an exception occurs. This guarantee makes the `finally` block an ideal location to place resource deallocation code for resources acquired and manipulated in the corresponding `try` block. If the `try` block executes successfully, the `finally` block executes immediately after the `try` block terminates. If an exception occurs in the `try` block, the `finally` block executes immediately after a `catch` handler completes exception handling. If the exception is not caught by a `catch` handler associated with that `try` block or if a `catch` handler associated with that `try` block throws an exception, the `finally` block executes, then the exception is processed by the next enclosing `try` block (if there is one).

Testing and Debugging Tip 11.3

A `finally` block typically contains code to release resources acquired in the corresponding `try` block; this makes the `finally` block an effective way to eliminate resource leaks.

Testing and Debugging Tip 11.4

The only reason a `finally` block will not execute if program control entered the corresponding `try` block is that the application terminates before `finally` can execute.

Performance Tip 11.3

As a rule, resources should be released as soon as it is apparent that they are no longer needed in a program, to make those resources immediately available for reuse, thus enhancing resource utilization in the program.

If one or more `catch` handlers follow a `try` block, the `finally` block is optional. If no `catch` handlers follow a `try` block, a `finally` block must appear immediately after the `try` block. If any `catch` handlers follow a `try` block, the `finally` block appears after the last `catch`. Only whitespace and comments can separate the blocks in a `try`/`catch`/`finally` sequence.

Common Programming Error 11.5

Placing the `finally` block before a `catch` handler is a syntax errors.

The C# application in Fig. 11.2 demonstrates that the `finally` block always executes, even if no exception occurs in the corresponding `try` block. The program consists of method `Main` (lines 10–59) and four other `static` methods that `Main` invokes to demonstrate `finally`—`DoesNotThrowException` (lines 62–85), `ThrowExceptionWithCatch` (lines 88–114), `ThrowExceptionWithoutCatch` (lines 117–138) and `ThrowExceptionCatchRethrow` (lines 141–173). [*Note:* We use `static` methods in this example so that `Main` can invoke these methods directly without creating any objects of class `UsingExceptions`. This enables us to concentrate on the mechanics of `try`/`catch`/`finally`.]

Line 14 of `Main` invokes method `DoesNotThrowException` (lines 62–85). The `try` block (lines 65–68) begins by outputting a message (line 67). The `try` block does not throw any exceptions, so program control reaches the closing brace of the `try` block and the `catch` handler (lines 71–74) and executes the `finally` block (lines 77–81) which outputs a message. At this point, program control continues with the first statement after the `finally` block (line 83), which outputs a message indicating that the end of the method has been reached. Then, program control returns to `Main`.

Line 20 of `Main` invokes method `ThrowExceptionWithCatch` (lines 88–114), which begins in its `try` block (lines 91–97) by outputting a message. Next, the `try` block creates a new `Exception` object and uses a ***`throw`*** *statement* to throw the exception object (lines 95–96). The `string` passed to the constructor becomes the exception object's error message. When a `throw` statement in a `try` block executes, the `try` block expires immediately, and program control continues at the first `catch` (lines 100–103) following this `try` block. In this example, the type thrown (`Exception`) matches the type specified in the `catch`, so line 102 outputs a message indicating the exception that occurred. Then, the `finally` block (lines 106–110) executes and outputs a message. At this point, program control continues with the first statement after the `finally` block (line 112), which outputs a message indicating that the end of the method has been reached, then program control returns to `Main`. Note, that in line 102, we use the exception object's `Message` property to access the error message associated with the exception—(the message passed to the `Exception` constructor). Section 11.6 discusses several properties of class `Exception`.

Common Programming Error 11.6

The expression of a ***`throw`***—*an exception object—must be of either class* ***`Exception`*** *or one of its derived classes.*

Lines 27–30 of `Main` define a `try` block in which `Main` invokes method `ThrowExceptionWithoutCatch` (lines 117–138). The `try` block enables `Main` to catch any exceptions thrown by `ThrowExceptionWithoutCatch`. The `try` block in lines 120–126 of `ThrowExceptionWithoutCatch` begins by outputting a message. Next, the `try` block throws an `Exception` (lines 124–125) and the `try` block expires immediately. Normally, program control would continue at the first `catch` following the `try` block. However, this `try` block does not have any corresponding `catch` handlers. Therefore, the exception is not caught in method `ThrowExceptionWithoutCatch`. Normal program control cannot continue until that exception is caught and processed. Thus, the CLR will terminate `ThrowExceptionWithoutCatch` and program control will return to `Main`. Before control returns to `Main`, the `finally` block (lines 129–133) executes and outputs a message. At this point, program control returns to **`Main`**—any statements appearing after the **`finally`** block would not execute. In this example, because the exception thrown at lines 127–128 is not caught: Method **`ThrowExceptionWithoutCatch`** always terminates after the **`finally`** block executes. In **`Main`**, the **`catch`** handler at lines 34–38 catches the exception and displays a message indicating that the exception was caught in **`Main`**.

```
 1  // Fig 11.2: UsingExceptions.cs
 2  // Using finally blocks.
 3
 4  using System;
 5
 6  // demonstrating that finally always executes
 7  class UsingExceptions
 8  {
 9     // entry point for application
10     static void Main( string[] args )
11     {
12        // Case 1: No exceptions occur in called method.
13        Console.WriteLine( "Calling DoesNotThrowException" );
14        DoesNotThrowException();
15
16        // Case 2: Exception occurs and is caught
17        // in called method.
18        Console.WriteLine( "\nCalling ThrowExceptionWithCatch" );
19        ThrowExceptionWithCatch();
20
21        // Case 3: Exception occurs, but not caught
22        // in called method, because no catch handlers.
23        Console.WriteLine(
24           "\nCalling ThrowExceptionWithoutCatch" );
25
26        // call ThrowExceptionWithoutCatch
27        try
28        {
29           ThrowExceptionWithoutCatch();
30        }
31
32        // process exception returned from
33        // ThrowExceptionWithoutCatch
34        catch
35        {
36           Console.WriteLine( "Caught exception from " +
37              "ThrowExceptionWithoutCatch in Main" );
38        }
39
40        // Case 4: Exception occurs and is caught
41        // in called method, then rethrown to caller.
42        Console.WriteLine(
43           "\nCalling ThrowExceptionCatchRethrow" );
44
```

Fig. 11.2 Demonstrating that **`finally`** blocks always execute regardless of whether or not an exception occurs. (Part 1 of 4.)

```
      // call ThrowExceptionCatchRethrow
      try
      {
         ThrowExceptionCatchRethrow();
      }

      // process exception returned from
      // ThrowExceptionCatchRethrow
      catch
      {
         Console.WriteLine( "Caught exception from " +
            "ThrowExceptionCatchRethrow in Main" );
      }

   } // end method Main

   // no exceptions thrown
   public static void DoesNotThrowException()
   {
      // try block does not throw any exceptions
      try
      {
         Console.WriteLine( "In DoesNotThrowException" );
      }

      // this catch never executes
      catch
      {
         Console.WriteLine( "This catch never executes" );
      }

      // finally executes because corresponding try executed
      finally
      {
         Console.WriteLine(
            "Finally executed in DoesNotThrowException" );
      }

      Console.WriteLine( "End of DoesNotThrowException" );

   } // end method DoesNotThrowException

   // throws exception and catches it locally
   public static void ThrowExceptionWithCatch()
   {
      // try block throws exception
      try
      {
         Console.WriteLine( "In ThrowExceptionWithCatch" );

```

Fig. 11.2 Demonstrating that `finally` blocks always execute regardless of whether or not an exception occurs. (Part 2 of 4.)

```
            throw new Exception(
               "Exception in ThrowExceptionWithCatch" );
         }

         // catch exception thrown in try block
         catch ( Exception error )
         {
            Console.WriteLine( "Message: " + error.Message );
         }

         // finally executes because corresponding try executed
         finally
         {
            Console.WriteLine(
               "Finally executed in ThrowExceptionWithCatch" );
         }

         Console.WriteLine( "End of ThrowExceptionWithCatch" );

      } // end method ThrowExceptionWithCatch

      // throws exception and does not catch it locally
      public static void ThrowExceptionWithoutCatch()
      {
         // throw exception, but do not catch it
         try
         {
            Console.WriteLine( "In ThrowExceptionWithoutCatch" );

            throw new Exception(
               "Exception in ThrowExceptionWithoutCatch" );
         }

         // finally executes because corresponding try executed
         finally
         {
            Console.WriteLine( "Finally executed in " +
               "ThrowExceptionWithoutCatch" );
         }

         // unreachable code; would generate logic error
         Console.WriteLine( "This will never be printed" );

      } // end method ThrowExceptionWithoutCatch

      // throws exception, catches it and rethrows it
      public static void ThrowExceptionCatchRethrow()
      {
         // try block throws exception
         try
         {
            Console.WriteLine( "In ThrowExceptionCatchRethrow" );
```

Fig. 11.2 Demonstrating that `finally` blocks always execute regardless of whether or not an exception occurs. (Part 3 of 4.)

```

         throw new Exception(
            "Exception in ThrowExceptionCatchRethrow" );
      }

      // catch any exception, place in object error
      catch ( Exception error )
      {
         Console.WriteLine( "Message: " + error.Message );

         // rethrow exception for further processing
         throw error;

         // unreachable code; would generate logic error
      }

      // finally executes because corresponding try executed
      finally
      {
         Console.WriteLine( "Finally executed in " +
            "ThrowExceptionCatchRethrow" );
      }

      // unreachable code; would generate logic error
      Console.WriteLine( "This will never be printed" );

   } // end method ThrowExceptionCatchRethrow

} // end class UsingExceptions
```

```
Calling DoesNotThrowException
In DoesNotThrowException
Finally executed in DoesNotThrowException
End of DoesNotThrowException

Calling ThrowExceptionWithCatch
In ThrowExceptionWithCatch
Message: Exception in ThrowExceptionWithCatch
Finally executed in ThrowExceptionWithCatch
End of ThrowExceptionWithCatch

Calling ThrowExceptionWithoutCatch
In ThrowExceptionWithoutCatch
Finally executed in ThrowExceptionWithoutCatch
Caught exception from ThrowExceptionWithoutCatch in Main

Calling ThrowExceptionCatchRethrow
In ThrowExceptionCatchRethrow
Message: Exception in ThrowExceptionCatchRethrow
Finally executed in ThrowExceptionCatchRethrow
Caught exception from ThrowExceptionCatchRethrow in Main
```

Fig. 11.2 Demonstrating that `finally` blocks always execute regardless of whether or not an exception occurs. (Part 4 of 4.)

Lines 46–49 of `Main` define a `try` block in which `Main` invokes method `ThrowExceptionCatchRethrow` (lines 141–173). The `try` block enables `Main` to catch any exceptions thrown by `ThrowExceptionCatchRethrow`. The `try` block in lines 144–150 of `ThrowExceptionCatchRethrow` begins by outputting a message. Next, the `try` block throws an `Exception` (lines 148–149). The `try` block expires immediately, and program control continues at the first `catch` (lines 153–161) following the `try` block. In this example, the type thrown (`Exception`) matches the type specified in the `catch`, so line 155 outputs a message indicating the exception that occurred. Line 158 uses the `throw` statement to *rethrow* the exception. This indicates that the `catch` handler performed partial processing (or no processing) of the exception and is now passing the exception back to the calling method (in this case `Main`) for further processing. Note that the expression to the `throw` statement is the reference to the exception that was caught. When rethrowing the original exception, you can also use the statement

```
throw;
```

with no expression. Section 11.6 discusses the `throw` statement with an expression. Such a `throw` statement enables programmers to catch an exception, create an exception object, then throw a different type of exception from the `catch` handler. Class library designers often do this to customize the exception types thrown from methods in their class libraries or to provide additional debugging information.

Software Engineering Observation 11.7

Before throwing an exception to a calling method, the method that throws the exception should release any resources acquired within the method before the exception occurred.[8]

Software Engineering Observation 11.8

Whenever possible, a method should handle exceptions that are thrown in that method, rather than passing the exceptions to another region of the program.

The exception handling in method `ThrowExceptionCatchRethrow` did not complete, because the program cannot run code in the `catch` handler placed after the invocation of the `throw` statement (line 158). Therefore, method `ThrowExceptionCatchRethrow` will terminate and return control to `Main`. Once again, the `finally` block (lines 164–168) will execute and output a message before control returns to `Main`. When control returns to `Main`, the `catch` handler at lines 53–57 catches the exception and displays a message indicating that the exception was caught. Then the program terminates.

Note that the point at which program control continues after the `finally` block executes depends on the exception-handling state. If the `try` block successfully completes or if a `catch` handler catches and handles an exception, control continues with the next statement after the `finally` block. If an exception is not caught or if a `catch` handler rethrows an exception, program control continues in the next enclosing `try` block. The enclosing `try` may be in the calling method or one of its callers. Nesting a `try`/`catch` sequence in a `try` block is also possible, in which case the outer `try` block's catch handlers would process any exceptions that were not caught in the inner `try`/`catch` sequence. If a `try` block has a corresponding `finally` block, the `finally` block executes even if the `try` block terminates due to a `return` statement; then the `return` occurs.

---

[8] "Best Practices for Handling Exceptions [C#]."

Common Programming Error 11.7

Throwing an exception from a `finally` *can be dangerous. If an uncaught exception is awaiting processing when the* `finally` *block executes and the* `finally` *block throws a new exception that is not caught in the* `finally` *block, the first exception is lost, and the new exception is the one passed to the next enclosing* `try` *block.*

Testing and Debugging Tip 11.5

When placing code that can throw an exception in a `finally` *block, always enclose that code in a* `try`*/*`catch` *sequence that catches the appropriate exception types. This prevents losing uncaught and rethrown exceptions that occur before the* `finally` *block executes.*

Software Engineering Observation 11.9

C#'s exception-handling mechanism removes error-processing code from the main line of a program to improve program clarity. Do not place `try`*-*`catch`*-*`finally` *around every statement that could throw an exception. Doing so makes programs difficult to read. Rather, place one* `try` *block around a significant portion of your code. Follow this* `try` *block with* `catch` *handlers that handle each of the possible exceptions and follow the* `catch` *handlers with a single* `finally` *block.*

11.6 `Exception` Properties

As we discussed in Section 11.4, exception data types derive from class **`Exception`**, which has several properties. These properties frequently are used to formulate error messages for a caught exception. Two important properties are ***`Message`*** and ***`StackTrace`***. Property **`Message`** stores the error message associated with an **`Exception`** object. This message may be a default message associated with the exception type or a customized message passed to an exception object's constructor when the exception object is constructed. Property **`StackTrace`** contains a **`string`** that represents the *method call stack*. The runtime environment keeps a list of method calls that have been made up to a given moment. The **`StackTrace`** **`string`** represents this sequential list of methods that had not finished processing at the time the exception occurred. The exact location at which the exception occurs in the program is called the exception's *throw point*.

Testing and Debugging Tip 11.6

A stack trace shows the complete method call stack at the time an exception occurred. This lets the programmer view the series of method calls that led to the exception. Information in the stack trace includes names of the methods on the call stack at the time of the exception, names of the classes in which those methods are defined, names of the namespaces in which those classes are defined and line numbers. The first line number in the stack trace indicates the throw point. Subsequent line numbers indicate the locations from which each method in the stack trace was called.

Another property used frequently by class library programmers is ***`InnerException`***. Typically, programmers use this property to "wrap" exception objects caught in their code, then throw new exception types that are specific to their libraries. For example, a programmer implementing an accounting system might have some account-number processing code in which account numbers are input as **`string`**s but represented with integers in the code. As you know, a program can convert **`string`**s to **`int`** values with **`Convert.ToInt32`**, which throws a **`FormatException`** when it encounters an invalid number format. When an invalid account-number format occurs, the accounting-system programmer might wish either to indicate an error message different from the default one supplied by **`FormatException`** or to indicate a new exception type, such as **`InvalidAccountNumberFormatException`**. In these cases, the programmer would provide code to catch the **`FormatException`**, then create an exception object in the **`catch`** handler, passing the original exception as one of the constructor arguments. The original exception object becomes the **`InnerException`** of the new exception object. When an **`InvalidAccountNumberFormatException`** occurs in code that uses the accounting-system library, the **`catch`** handler that catches the exception can view the original exception via the property **`InnerException`**. Thus, the exception indicates that an invalid account number was specified and that the particular problem was an invalid number format.

Our next example (Fig. 11.3) demonstrates properties **`Message`**, **`StackTrace`** and **`InnerException`** and method **`ToString`**. In addition, this example demonstrates *stack unwinding*—the process that attempts to locate an appropriate **`catch`** handler for an uncaught exception. As we discuss this example, we keep track of the methods on the call stack, so we can discuss property **`StackTrace`** and the stack-unwinding mechanism.

Program execution begins with the invocation of **`Main`**, which becomes the first method on the method call stack. Line 16 of the **`try`** block in **`Main`** invokes **`Method1`** (lines 43–46), which becomes the second method on the stack. If **`Method1`** throws an exception, the **`catch`** handler at lines 22–38 handle the exception and output information about the exception that occurred. Line 45 of **`Method1`** invokes **`Method2`** (lines 49–52), which becomes the third method on the stack. Then, line 51 of **`Method2`** invokes **`Method3`** (defined at lines 55–70) which becomes the fourth method on the stack.

Testing and Debugging Tip 11.7

When reading a stack trace, start from the top of the stack trace and read the error message first. Then, read the remainder of the stack trace, looking for the first line that indicates code that you wrote in your program. Normally, this is the location that caused the exception.

```
1   // Fig 11.3: Properties.cs
2   // Stack unwinding and Exception class properties.
3
4   using System;
5
6   // demonstrates using the Message, StackTrace and
7   // InnerException properties
8   class Properties
9   {
10     static void Main( string[] args )
11     {
12        // call Method1, any Exception it generates will be
13        // caught in the catch handler that follows
14        try
15        {
16           Method1();
17        }
```

Fig. 11.3 **`Exception`** properties and stack unwinding. (Part 1 of 3.)

```

      // Output string representation of Exception, then
      // output values of InnerException, Message,
      // and StackTrace properties
      catch ( Exception exception )
      {
         Console.WriteLine(
            "exception.ToString(): \n{0}\n",
            exception.ToString() );

         Console.WriteLine( "exception.Message: \n{0}\n",
            exception.Message );

         Console.WriteLine( "exception.StackTrace: \n{0}\n",
            exception.StackTrace );

         Console.WriteLine(
            "exception.InnerException: \n{0}",
            exception.InnerException );

      } // end catch

   } // end Main

   // calls Method2
   public static void Method1()
   {
      Method2();
   }

   // calls Method3
   public static void Method2()
   {
      Method3();
   }

   // throws an Exception containing an InnerException
   public static void Method3()
   {
      // attempt to convert non-integer string to int
      try
      {
         Convert.ToInt32( "Not an integer" );
      }

      // catch FormatException and wrap it in new Exception
      catch ( FormatException error )
      {
         throw new Exception(
            "Exception occurred in Method3", error );
      }

   } // end method Method3
```

Fig. 11.3 `Exception` properties and stack unwinding. (Part 2 of 3.)

```

} // end class UsingExceptions
```

```
exception.ToString():
System.Exception: Exception occurred in Method3 --->
   System.FormatException: Input string was not in a correct format.
   at System.Number.ParseInt32(String s, NumberStyles style,
      NumberFormatInfo info)
   at System.Convert.ToInt32(String s)
   at Properties.Method3() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 60
   --- End of inner exception stack trace ---
   at Properties.Method3() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 66
   at Properties.Method2() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 51
   at Properties.Method1() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 45
   at Properties.Main(String[] args) in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 16

exception.Message:
Exception occurred in Method3

exception.StackTrace:
   at Properties.Method3() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 66
   at Properties.Method2() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 51
   at Properties.Method1() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 45
   at Properties.Main(String[] args) in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 16

exception.InnerException:
System.FormatException: Input string was not in a correct format.
   at System.Number.ParseInt32(String s, NumberStyles style,
      NumberFormatInfo info)
   at System.Convert.ToInt32(String s)
   at Properties.Method3() in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\
         properties\properties.cs:line 60
```

Fig. 11.3 `Exception` properties and stack unwinding. (Part 3 of 3.)

At this point, the method call stack for the program is

```
Method3
Method2
Method1
Main
```

with the last method called (`Method3`) at the top and the first method called (`Main`) at the bottom. The `try` block (lines 58–61) in `Method3` invokes method `Convert.ToInt32` (line 60) and attempts to convert a `string` to an `int`. At this point, `Convert.ToInt32` becomes the fifth and final method on the call stack.

The argument to `Convert.ToInt32` is not in integer format, so line 60 throws a `FormatException` that is caught at line 64 in `Method3`. The exception terminates the call to `Convert.ToInt32`, so the method is removed from the method call stack. The `catch` handler creates an `Exception` object, then throws it. The first argument to the `Exception` constructor is the custom error message for our example, "`Exception occurred in Method3`." The second argument is the `InnerException` object—the `FormatException` that was caught. Note that the `StackTrace` for this new exception object will reflect the point at which the exception was thrown (line 66). Now, `Method3` terminates, because the exception thrown in the `catch` handler is not caught in the method body. Thus, control will be returned to the statement that invoked `Method3` in the prior method in the call stack (`Method2`). This removes or *unwinds* `Method3` from the method-call stack.

Good Programming Practice 11.2

When catching and rethrowing an exception, provide additional debugging information in the rethrown exception. To do so, create an `Exception` *object with more specific debugging information and pass the original caught exception to the new exception object's constructor to initialize the* `InnerException` *property.*[9]

When control returns to line 51 in `Method2`, the CLLR determines that line 51 is not in a `try` block. Therefore, the exception cannot be caught in `Method2`, and `Method2` terminates. This unwinds `Method2` from the method-call stack and returns control to line 45 in `Method1`. Here again, line 45 is not in a `try` block, so the exception cannot be caught in `Method1`. The method terminates and unwinds from the call stack, returning control to line 16 in `Main`, which is in a `try` block. The `try` block in `Main` expires, and the `catch` handler at lines (22–38) catches the exception. The `catch` handler uses method `ToString` and properties `Message`, `StackTrace` and `InnerException` to produce the output. Note that stack unwinding continues until either a `catch` handler catches the exception or the program terminates.

The first block of output (reformatted for readability) in Fig. 11.3 shows the exception's `string` representation returned from method `ToString`. This begins with the name of the exception class followed by the `Message` property value. The next eight lines show the `string` representation of the `InnerException` object. The remainder of that block of output shows the `StackTrace` for the exception thrown in `Method3`. Note that the `StackTrace` represents the state of the method-call stack at the throw point of the exception, not at the point where the exception eventually is caught. Each of the `StackTrace` lines that begins with "`at`" represents a method on the call stack. These lines indicate the method in which the exception occurred, the file in which that method resides and the line number in the file. Also, note that the stack trace includes the inner-exception stack trace.

---

[9] "Best Practices for Handling Exceptions [C#]," *.NET Framework Developer's Guide*, Visual Studio .NET Online Help.

Testing and Debugging Tip 11.8

When catching and rethrowing an exception, provide additional debugging information in the rethrown exception. To do so, create an `Exception` object containing more specific debugging information and then pass the original caught exception to the new exception object's constructor to initialize the `InnerException` property.

Method `ToString` of an exception returns a `string` containing the name of the exception, the optional character `string` supplied when the exception was constructed, the inner exception (if there is one) and a stack trace.

The next block of output (two lines) simply displays the `Message` property (`Exception occurred in Method3`) of the exception thrown in `Method3`.

The third block of output displays the `StackTrace` property of the exception thrown in `Method3`. Note that the `StackTrace` property includes the stack trace starting from line 66 in `Method3`, because that is the point at which the `Exception` object was created and thrown. The stack trace always begins from the exception's throw point.

Finally, the last block of output displays the `ToString` representation of the `InnerException` property, which includes the namespace and class names of that exception object, its `Message` property and its `StackTrace` property.

11.7 Programmer-Defined Exception Classes

In many cases, programmers can use existing exception classes from the .NET Framework to indicate exceptions that occur in their programs. However, in some cases, programmers may wish to create exception types that are more specific to the problems that occur in their programs. *Programmer-defined exception classes* should derive directly or indirectly from class `ApplicationException` of namespace `System`.

Good Programming Practice 11.3

Associating each type of malfunction with an appropriately named exception class improves program clarity.

Software Engineering Observation 11.10

Before creating programmer-defined exception classes, investigate the existing exception classes in the .NET Framework to determine whether an appropriate exception type already exists.

Software Engineering Observation 11.11

Programmers should create exception classes only if they need to catch and handle the new exceptions differently from other existing exception types.

Figure 11.5 and Fig. 11.5 demonstrate defining and using a programmer-defined exception class. Class `NegativeNumberException` (Fig. 11.4) is a programmer-defined exception class representing exceptions that occur when a program performs an illegal operation on a negative number, such as the square root of a negative number.

```
// Fig 11:4: NegativeNumberException.cs
// NegativeNumberException represents exceptions caused by illegal
// operations performed on negative numbers

using System;

// NegativeNumberException represents exceptions caused by
// illegal operations performed on negative numbers
class NegativeNumberException : ApplicationException
{
   // default constructor
   public NegativeNumberException()
      : base( "Illegal operation for a negative number" )
   {
   }

   // constructor for customizing error message
   public NegativeNumberException( string message )
      : base( message )
   {
   }

   // constructor for customizing error message and
   // specifying inner exception object
   public NegativeNumberException(
      string message, Exception inner )
         : base( message, inner )
   {
   }

} // end class NegativeNumberException
```

Fig. 11.4 `ApplicationException` subclass thrown when a program performs illegal operations on negative numbers.

According to Microsoft,[10] programmer-defined exceptions should extend class **`ApplicationException`**, should have a class name that ends with “Exception” and should define three constructors—a default constructor, a constructor that receives a **`string`** argument (the error message) and a constructor that receives a **`string`** argument and an **`Exception`** argument (the error message and the inner-exception object).

`NegativeNumberException`s most likely occur during arithmetic operations, so it seems logical to derive class **`NegativeNumberException`** from class **`ArithmeticException`**. However, class **`ArithmeticException`** derives from class **`SystemException`**—the category of exceptions thrown by the CLR. **`ApplicationException`** specifically is the base class for exceptions thrown by a user program, not by the CLR.

Class **`SquareRootTest`** (Fig. 11.5) demonstrates our programmer-defined exception class. The application enables the user to input a numeric value, then invokes method **`SquareRoot`** (lines 42–52) to calculate the square root of that value. For this purpose, **`SquareRoot`** invokes class **`Math`**’s ***`Sqrt`*** method, which receives a nonnegative **`double`** value as its argument. If the argument is negative, method **`Sqrt`** normally returns constant **`NaN`** from class **`Double`**. In this program, we would like to prevent the user from calculating the square root of a negative number. If the numeric value received from the user is negative, **`SquareRoot`** throws a **`NegativeNumberException`** (lines 46–47). Otherwise, **`SquareRoot`** invokes class **`Math`**’s ***`Sqrt`*** method to compute the square root.

---

[10] “Best Practices for Handling Exceptions [C#],” *.NET Framework Developer's Guide*, Visual Studio .NET Online Help.

When the user inputs a value and clicks the **Square Root** button, the program invokes method **`squareRootButton_Click`** (lines 56–85). The **`try`** block (lines 62–68) attempts to invoke **`SquareRoot`** with the value input by the user. If the user input is not a valid number, a **`FormatException`** occurs, and the **`catch`** handler at lines 71–76 processes the exception. If the user inputs a negative number, method **`SquareRoot`** throws a **`NegativeNumberException`** (lines 46–47). The **`catch`** handler at lines 79–83 catches and handles that exception.

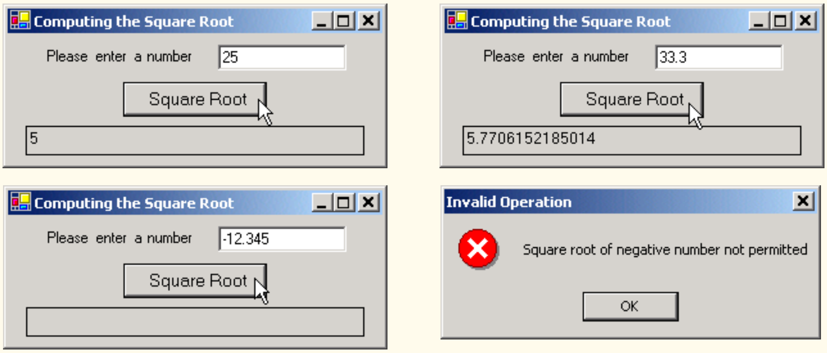
```
 1  // Fig 11.5: SquareRootTest.cs
 2  // Demonstrating a programmer-defined exception class.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  // accepts input and computes the square root of that input
12  public class SquareRootTest : System.Windows.Forms.Form
13  {
14     private System.Windows.Forms.Label inputLabel;
15     private System.Windows.Forms.TextBox inputTextBox;
16
17     private System.Windows.Forms.Button squareRootButton;
18
19     private System.Windows.Forms.Label outputLabel;
20
21     // Required designer variable.
22     private System.ComponentModel.Container components = null;
23
24     // default constructor
25     public SquareRootTest()
26     {
27        // Required for Windows Form Designer support
28        InitializeComponent();
29     }
30
31     // Visual Studio .NET generated code
32
```

Fig. 11.5 **`SquareRootTest`** class thrown an exception if error occurs when calculating the square root. (Part 1 of 3.)

```
   // main entry point for the application
   [STAThread]
   static void Main()
   {
      Application.Run( new SquareRootTest() );
   }

   // computes the square root of its parameter; throws
   // NegativeNumberException if parameter is negative
   public double SquareRoot( double operand )
   {
      // if negative operand, throw NegativeNumberException
      if ( operand < 0 )
         throw new NegativeNumberException(
            "Square root of negative number not permitted" );

      // compute the square root
      return Math.Sqrt( operand );

   } // end class SquareRoot

   // obtain user input, convert to double and calculate
   // square root
   private void squareRootButton_Click(
      object sender, System.EventArgs e )
   {
      outputLabel.Text = "";

      // catch any NegativeNumberExceptions thrown
      try
      {
         double result =
            SquareRoot( Double.Parse( inputTextBox.Text ) );

         outputLabel.Text = result.ToString();
      }

      // process invalid number format
      catch ( FormatException notInteger )
      {
         MessageBox.Show( notInteger.Message,
            "Invalid Operation", MessageBoxButtons.OK,
            MessageBoxIcon.Error );
      }

      // display MessageBox if negative number input
      catch ( NegativeNumberException error )
      {
         MessageBox.Show( error.Message, "Invalid Operation",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      }

```

Fig. 11.5 `SquareRootTest` class thrown an exception if error occurs when calculating the square root. (Part 2 of 3.)

```
      } // end method squareRootButton_Click

   } // end class SquareRootTest
```



Computing the Square Root

Please enter a number: 25

5

Computing the Square Root

Please enter a number: 33.3

5.7706152185014

Computing the Square Root

Please enter a number: -12.345

Invalid Operation

Square root of negative number not permitted

Fig. 11.5 `SquareRootTest` class thrown an exception if error occurs when calculating the square root. (Part 3 of 3.)

11.8 Handling Overflows with Operators `checked` and `unchecked`

In .NET, the primitive data types are stored in fixed-size structures. For instance, the maximum value of an **`int`** is 2,147,483,647. In integer arithmetic, a value larger than 2,147,483,647 causes *overflow*—type **`int`** cannot represent such a number. Overflow also can occur with other C# primitive types. Overflows often cause programs to produce incorrect results.

C# provides operators ***`checked`*** and ***`unchecked`*** to specify whether integer arithmetic occurs in a *checked context* or *unchecked context*. In a checked context, the CLR throws an **`OverflowException`** (namespace **`System`**) if overflow occurs during evaluation of an arithmetic expression. In an unchecked context, the result is truncated if overflow occurs.

The operators **`++`**, **`--`**, **`*`**, **`/`**, **`+`** and **`-`** (both unary and binary) may cause overflow when used with integral data types (such as **`int`** and **`long`**). Also, explicit conversions between integral data types can cause overflow. For example, converting the integer 1,000,000 from **`int`** to **`short`** results in overflow, because a **`short`** can store a maximum value of 32,767. Figure 11.6 demonstrates overflows occurring in both checked and unchecked contexts.

The program begins by defining **`int`** variables **`number1`** and **`number2`** (lines 11–12) and assigning each variable the maximum value for an **`int`**—2,147,483,647 (defined by **`Int32.MaxValue`**). Next, line 13 defines variable **`sum`** (initialized to 0) to store the sum of **`number1`** and **`number2`**. Then, lines 15–16 output the values of **`number1`** and **`number2`**.

Lines 19–25 define a **`try`** block in which line 24 adds **`number1`** and **`number2`** in a checked context. The expression to evaluate in a checked context appears in parentheses following keyword **`checked`**. Variables **`number1`** and **`number2`** already contain the maximum value for an **`int`**, so adding these values causes an **`OverflowException`**. The **`catch`** handler at lines 28–31 catches the exception and outputs its **`string`** representation.

Line 39 performs the same calculation in an unchecked context. The result of the calculation should be 4,294,967,294. However, this value requires more memory than an **`int`** can store, so operator **`unchecked`** truncates part of the value, resulting in **`-2`** in the output. As you can see, the result of the unchecked calculation is not the actual sum of the variables.

```
 1  // Fig 11.6: Overflow.cs
 2  // Demonstrating operators checked and unchecked.
 3
 4  using System;
 5
 6  // demonstrates using the checked and unchecked operators
 7  class Overflow
 8  {
 9     static void Main( string[] args )
10     {
11        int number1 = Int32.MaxValue;  // 2,147,483,647
12        int number2 = Int32.MaxValue;  // 2,147,483,647
13        int sum = 0;
14
15        Console.WriteLine(
16           "number1: {0}\nnumber2: {1}", number1, number2 );
17
18        // calculate sum of number1 and number2
19        try
20        {
21           Console.WriteLine(
22              "\nSum integers in checked context:" );
23
24           sum = checked( number1 + number2 );
25        }
26
27        // catch overflow exception
28        catch ( OverflowException overflowException )
29        {
30           Console.WriteLine( overflowException.ToString() );
31        }
32
33        Console.WriteLine(
34           "\nsum after checked operation: {0}", sum );
35
36        Console.WriteLine(
37           "\nSum integers in unchecked context:" );
38
39        sum = unchecked( number1 + number2 );
40
41        Console.WriteLine(
42           "sum after unchecked operation: {0}", sum );
43
44     } // end method Main
45
46  } // end class Overflow
```

Fig. 11.6 Operators **`checked`** and **`unchecked`** and the handling of arithmetic overflow. (Part 1 of 2.)

```
number1: 2147483647
number2: 2147483647

Sum integers in checked context:
System.OverflowException: Arithmetic operation resulted in an overflow.
   at Overflow.Overflow.Main(String[] args) in
      f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_09\
         overflow\overflow.cs:line 24

sum after checked operation: 0

Sum integers in unchecked context:
sum after unchecked operation: -2
```

Fig. 11.6 Operators `checked` and `unchecked` and the handling of arithmetic overflow. (Part 2 of 2.)

By default, calculations occur in an unchecked context—a dangerous practice, unless the calculations are preformed on constant expressions (such as literal integer values). Constant expressions are evaluated in a checked context at compile time. Overflows in such expressions results in compile time errors. It is possible to specify in a project's properties that the default context for evaluating non-constant expressions should be to check for arithmetic overflow. In the properties for your project, you can set the checked context as the default. To do so, first select your project in the **Solution Explorer**. Next, in the **View** menu, select **Property Pages**. In the **Property Pages** dialog, click the **Configuration Properties** folder. Under **Code Generation**, change the value of **Check for Arithmetic Overflow/Underflow** to `true`.

Good Programming Practice 11.4

Use a checked context when performing calculations that can result in overflows. The programmer should define exception handlers that can process the overflow.

Software Engineering Observation 11.12

Keywords `checked` and `unchecked` can evaluate blocks of statements in checked or unchecked contexts by following the appropriate keyword with a block of code in braces (`{}`).

In this chapter, we demonstrated how the exception-handling mechanism works and discussed how to make applications more robust by writing exception handlers to process potential problems. As programmers develop new applications, it is important to investigate potential exceptions thrown by the methods your program invokes or by the CLR, then implement appropriate exception-handling code to make those applications more robust.

SUMMARY

- An exception is an indication of a problem that occurs during a program's execution.
- Exception handling enables programmers to create applications that can resolve exceptions, often allowing a program to continue execution as if no problems were encountered.
- Exception handling enables programmers to write clear, robust and more fault-tolerant programs.
- Exception handling enables the programmer to remove error-handling code from the "main line" of the program's execution. This improves program clarity and enhances modifiability.

- Exception handling is designed to process synchronous errors, such as out-of-range array subscripts, arithmetic overflow, division by zero, invalid method parameters and memory exhaustion.
- Exception handling is not designed to process asynchronous events, such as disk-I/O completions, network-message arrivals, mouse clicks and keystrokes.
- When a method detects an error and is unable to handle it, the method throws an exception. There is no guarantee that there will be an exception handler to process that kind of exception. If there is, the exception will be caught and handled.
- In debug mode, when the program detects an uncaught exception, a dialog box appears that enables the programmer to view the problem in the debugger or continue program execution by ignoring the problem that occurred.
- A **`try`** block consists of keyword **`try`** followed by braces (**`{}`**) that delimit a block of code in which exceptions could occur.
- Immediately following the **`try`** block are zero or more **`catch`** handlers. Each **`catch`** specifies in parentheses an exception parameter representing the exception type the **`catch`** can handle.
- If an exception parameter includes an optional parameter name, the **`catch`** handler can use that parameter name to interact with a caught exception object.
- There can be one parameterless **`catch`** handler that catches all exception types.
- After the last **`catch`** handler, an optional **`finally`** block contains code that always executes, regardless of whether an exception occurs.
- When a method called in a program or the CLR detects a problem, the method or CLR throws an exception. The point in the program at which an exception occurs is called the throw point.
- Exceptions are objects of classes that inherit directly or indirectly from class **`Exception`**.
- C# uses the termination model of exception handling. If an exception occurs in a **`try`** block, the block expires and program control transfers to the first **`catch`** handler following the **`try`** block.
- The CLR searches for the first **`catch`** handler that can process the type of exception that occurred. The appropriate handler is the first one in which the thrown exception's type matches, or is derived from, the exception type specified by the **`catch`** handler's exception parameter.
- If no exceptions occur in a **`try`** block, the CLR ignores the exception handlers for that block.
- If no exceptions occur or if an exception is caught and handled, the program resumes execution with the next statement after the **`try`**/**`catch`**/**`finally`** sequence.
- If an exception occurs in a statement that is not in a **`try`** block, the method containing that statement terminates immediately—a process called stack unwinding.
- When a **`try`** block terminates, local variables defined in the block go out of scope.
- If the argument to **`Convert.ToInt32`** is not an integer, a **`FormatException`** occurs.
- In integer arithmetic, an attempt to divide by zero causes a **`DivideByZeroException`**.
- A **`try`** block encloses the code that could throw exceptions and the code that should not execute if an exception occurs.
- Each **`catch`** handler begins with keyword **`catch`** followed by an optional exception parameter that specifies the type of exception handled by the **`catch`** handler. The exception-handling code appears in the body of the **`catch`** handler.
- Only the matching **`catch`** handler executes if an exception occurs. When program control reaches the closing brace of a **`catch`** handler, the CLR considers the exception handled, and program control continues with the first statement after the **`try`**/**`catch`** sequence.

- If a **`catch`** handler specifies an exception type and an exception parameter name, the exception handler's body can interact with the caught exception object. The exception parameter can be omitted if the exception handler does not require access to the exception object's properties.
- The exception-handling mechanism allows only objects of class **`Exception`** and its derived classes to be thrown and caught. Class **`Exception`** of namespace **`System`** is the base class of the .NET Framework exception hierarchy.
- **`ApplicationException`** is a base class programmers can extend to create new exception data types that are specific to their applications. Programs can recover from most **`ApplicationException`**s and continue execution.
- The Common Language Runtime generates **`SystemException`**s. If a program attempts to access an out-of-range array subscript, the CLR throws an **`IndexOutOfRangeException`**. Attempting to manipulate an object through a **`null`** reference causes a **`NullReferenceException`**.
- Programs typically cannot recover from most exceptions thrown by the CLR. Therefore, programs generally should not throw **`SystemException`**s nor attempt to catch.
- A **`catch`** handler can catch exceptions of a particular type or can use a base-class type to catch exceptions in a hierarchy of related exception types. A **`catch`** handler that specifies an exception parameter of type **`Exception`** can catch all exceptions, because **`Exception`** is the base class of all exception classes.
- For methods in the .NET Framework classes, you should look at the detailed description of the method in the online documentation to determine whether that method throws exceptions.
- Information on exceptions thrown by the CLR appears in the *C# Language Specification*, which is located in the online documentation.
- Many computer operating systems prevent more than one program from manipulating a resource at the same time. Therefore, when a program no longer needs a resource, the program normally releases the resource to allow other programs to use the resource. This helps prevent resource leaks, and helps ensure that resources are available when needed.
- In C and C++, the most common resource leaks are memory leaks that occur when a program allocates memory, but does not deallocate the memory when the memory is no longer needed in the program. The Common Language Runtime performs garbage collection of memory no longer needed by an executing program, thus avoiding such memory leaks.
- A program should release a resource when the resource is no longer needed. The **`finally`** block is guaranteed to execute if program control enters the corresponding **`try`** block, regardless of whether that **`try`** block executes successfully or an exception occurs. This guarantee makes the **`finally`** block an ideal location to place resource-deallocation code for resources acquired and manipulated in the corresponding **`try`** block.
- If one or more **`catch`** handlers follow a **`try`** block, the **`finally`** block is optional. If no **`catch`** handlers follow a **`try`** block, a **`finally`** block must appear immediately after the **`try`** block. If any **`catch`** handlers follow a **`try`** block, the **`finally`** block appears after the last **`catch`**.
- Only whitespace and comments can separate the blocks in a **`try`**/**`catch`**/**`finally`** sequence.
- A **`throw`** statement throws an exception object.
- A **`throw`** statement can be used in a **`catch`** handler to rethrow an exception. This indicates that the **`catch`** handler performed partial processing of the exception and is now passing the exception back to a calling method for further processing.
- **`Exception`** property **`Message`** stores the error message associated with an **`Exception`** object. This message may be a default message associated with the exception type or a customized message passed to an exception object's constructor at the time a program creates the exception.

- **Exception** property **StackTrace** contains a string that represents the method-call stack at the throw point of the exception.
- **Exception** property **InnerException** typically is used to “wrap” a caught exception object in a new exception object, then throw the object of that new exception type.
- When an exception is uncaught in a method, the method terminates. This removes or unwinds the method from the method-call stack.
- Programmer-defined exceptions should extend class **ApplicationException**, should have a class name that ends with “Exception” and should define a default constructor, a constructor that receives a **string** argument (the error message) and a constructor that receives a **string** argument and an **Exception** argument (the error message and the inner-exception object).
- Overflow occurs in integer arithmetic when the value of an expression is greater than the maximum value that can be stored in a particular integral data type.
- C# provides operators **checked** and **unchecked** to specify whether arithmetic occurs in a checked context or an unchecked context. In a checked context, operator **checked** throws an **OverflowException** if overflow occurs when evaluating an arithmetic expression. In an unchecked context, operator **unchecked** truncates the result if overflow occurs (normally, a dangerous thing to allow).
- The operators **++**, **--**, *****, **/**, **+** and **-** (both unary and binary) can cause overflow when used with integral data types (such as **int** and **long**). Also, explicit conversions between integral data types can cause overflow.
- The expression that is to be evaluated in a checked or unchecked context appears in parentheses following keyword **checked** or **unchecked**, respectively. Also, entire blocks of code can execute in a checked or unchecked context by placing keyword checked or unchecked before the opening left brace of the block.
- By default, calculations are performed in the unchecked context.

TERMINOLOGY

ApplicationException class
arithmetic overflow
asynchronous event
C# Language Specification
call stack
catch all exception types
catch block (or handler)
checked context
checked operator
Common Language Runtime (CLR)
disk I/O completion
divide by zero
DivideByZeroException class
DivideByZeroTest.cs
Double class
eliminate resource leaks
error-processing code
exception
Exception class
exception handler
fault-tolerant program
finally block
FormatException class
Handling a divide-by-zero exception
IndexOutOfRangeException class
inheritance with exceptions
InnerException property of **Exception**
integral data types
Koenig, Andrew
MaxValue constant of **Int32**
memory leak
Message
Message property of class **Exception**
Message property of **Exception**
method call stack
NaN constant of class **Double**
negative infinity
network message arrival
NullReferenceException
out-of-range array subscript
overflow
OverflowException class

polymorphic processing of related errors
positive infinity
release resource
resource leak
result of an uncaught exception
resumption model of exception handling
rethrow an exception
robust application
run-time exception
`Sqrt` method of `Math`
`SquareRootTest.cs`
stack unwinding
`StackTrace` property of `Exception`
Stroustrup, Bjarne
synchronous error
`SystemException` class
termination model of exception handling
throw an exception
throw point
`throw` statement
`ToInt32` method of `Convert`
`ToString`
`try` block
`try` block expires
unchecked context
`unchecked` operator
programmer-defined exception classes

SELF-REVIEW EXERCISES

11.1 Fill in the blanks in each of the following statements:

a) Exception handling deals with __________ errors, but not __________ errors.
b) A method __________ an exception when that method detects that a problem occurred.
c) The __________ block associated with a `try` block always executes.
d) Exception objects are derived from class __________.
e) The statement that throws an exception is called the __________ of the exception.
f) A __________ block encloses code that could throw an exception.
g) If the catch-all exception handler is declared before another exception handler, a __________ occurs.
h) An uncaught exception in a method causes that method to __________ from the method-call stack.
i) Method `Convert.ToInt32` can throw a __________ exception if its argument is not a valid integer value.
j) Runtime exceptions derive from class __________.
k) To force an exception to occur when arithmetic overflow occurs in integer arithmetic, use operator __________.

11.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Exceptions always are handled in the method that initially detects the exception.
b) Programmer-defined exception classes should extend class `SystemException`.
c) Accessing an out-of-bounds array subscript causes the CLR to throw an exception.
d) A `finally` block is optional after a `try` block.
e) If a `finally` block appears in a method, that `finally` block is guaranteed to execute.
f) Returning to the throw point of an exception using keyword `return` is possible.
g) Exceptions can be rethrown.
h) The `checked` operator causes a syntax error when integral arithmetic overflow occurs.
i) Property `Message` returns a `string` indicating the method from which the exception was thrown.
j) Exceptions can be thrown only by methods explicitly called in a `try` block.

ANSWERS TO SELF-REVIEW EXERCISES

11.1 a) synchronous, asynchronous. b) throws. c) `finally`. d) `Exception`. e) throw point. f) `try`. g) syntax error. h) unwind. i) `FormatException`. j) `SystemException`. k) `checked`.

11.2 a) False. Exceptions are handled by calling methods on the method-call stack. b) False. Programmer-defined exception classes should extend class **`ApplicationException`**. c) True. d) False. The **`finally`** block is option *only* if there is at least one **`catch`** handler. If there are not **`catch`** handlers, the **`finally`** block is *required*. e) False. The **`finally`** block will execute only if program control entered the corresponding **`try`** block. f) False. Keyword **`return`** causes control to **`return`** to the caller. g) True. h) False. The **`checked`** operator causes an exception when arithmetic overflow occurs at execution time. i) False. Property **`Message`** returns a **`string`** representing the error message. j) False. Exceptions can be thrown by any method, called from a **`try`** block or not. Also, the CLR can throw exceptions.

EXERCISES

11.3 Use inheritance to create an exception base class and various exception-derived classes. Write a program to demonstrate that the **`catch`** specifying the base class catches derived-class exceptions.

11.4 Write a C# program that demonstrates how various exceptions are caught with

```
catch ( Exception exception )
```

11.5 Write a C# program that shows the importance of the order of exception handlers. Write two programs: One with the correct order of **`catch`** handlers, and one with an incorrect order (i.e., place the base class exception handler before the derived-class exception handlers). Show that if you attempt to catch a base-class exception type before a derived-class exception type, the derived-class exceptions are not invoked (which potentially yield logical errors in routine). Explain why these errors occur.

11.6 Exceptions can be used to indicate problems that occur when an object is being constructed. Write a C# program that shows a constructor passing information about constructor failure to an exception handler that occurs after a **`try`** block. The exception thrown also should contain the arguments sent to the constructor.

11.7 Write a C# program that demonstrates rethrowing an exception.

11.8 Write a C# program that shows that a method with its own **`try`** block does not have to **`catch`** every possible exception that occurs within the **`try`** block. Some exceptions can slip through to, and be handled in, other scopes.

12

Graphical User Interface Concepts: Part 1

Objectives

- To understand the design principles of graphical user interfaces.
- To understand, use and create events.
- To understand the namespaces containing graphical user interface components and event-handling classes and interfaces.
- To be able to create graphical user interfaces.
- To be able to create and manipulate buttons, labels, lists, textboxes and panels.
- To be able to use mouse and keyboard events.

… the wisest prophets make sure of the event first.
Horace Walpole

...The user should feel in control of the computer; not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.
Inside Macintosh, Volume 1
Apple Computer, Inc. 1985

All the better to see you with my dear.
The Big Bad Wolf to Little Red Riding Hood

Outline

12.1 Introduction
12.2 Windows Forms
12.3 Event-Handling Model
  12.3.1 Basic Event Handling
12.4 Control Properties and Layout
12.5 `Labels`, `TextBoxes` and `Buttons`
12.6 `GroupBoxes` and `Panels`
12.7 `CheckBoxes` and `RadioButtons`
12.8 `PictureBoxes`
12.9 Mouse Event Handling
12.10 Keyboard Event Handling

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

12.1 Introduction

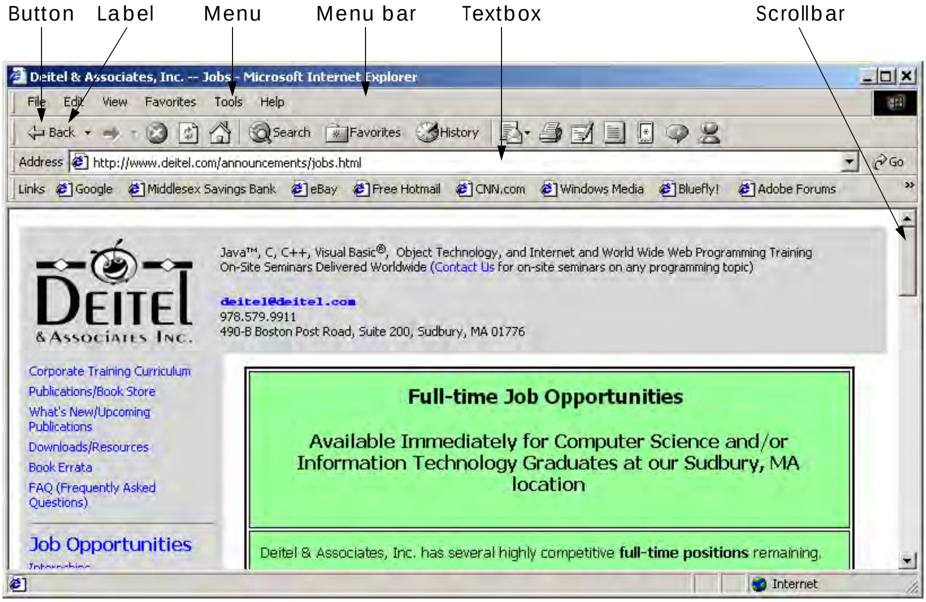
A *graphical user interface* (*GUI*) allows users to interact with a program visually. A GUI (pronounced "GOO-EE") gives a program a distinctive "look" and "feel." By providing different applications with a consistent set of intuitive user-interface components, GUIs allow users to spend less time trying to remember which keystroke sequences perform what functions and spend more time using the program in a productive manner.

Look-and-Feel Observation 12.1

Consistent user interfaces enable users to learn new applications faster.

As an example of a GUI, Fig. 12.1 contains an Internet Explorer window with some of its *GUI components* labeled. In the window, there is a *menu bar* containing *menus*, including **File**, **Edit**, **View**, **Favorites**, **Tools** and **Help**. Below the menu bar is a set of *buttons*; each has a defined task in Internet Explorer. Below the buttons is a *textbox*, in which the user can type the location of a World Wide Web site to visit. To the left of the textbox is a *label* that indicates the textbox's purpose. On the far right and bottom there are *scrollbars*. Scrollbars are used when there is more information in a window than can be displayed at once. By moving the scrollbars back and forth, the user can view different portions of the Web page. The menus, buttons, textboxes, labels and scrollbars are part of Internet Explorer's GUI. They form a user-friendly interface through which the user interacts with the Internet Explorer Web browser.

GUIs are built from GUI components (sometimes called *controls* or *widgets*—short for *window gadgets*). A GUI component is an object with which the user interacts via the mouse or keyboard. Several common GUI components are listed in Fig. 12.2. In the sections that follow, we discuss each of these GUI components in detail. In the next chapter, we discuss more advanced GUI components.


Button
Label
Menu
Menu bar
Textbox
Scrollbar
Deitel & Associates, Inc. -- Jobs - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.deitel.com/announcements/jobs.html
Go
Links Google Middlesex Savings Bank eBay Free Hotmail CNN.com Windows Media Bluefly! Adobe Forums
Deitel & Associates Inc.
Corporate Training Curriculum
Publications/Book Store
What's New/Upcoming Publications
Downloads/Resources
Book Errata
FAQ (Frequently Asked Questions)
Job Opportunities
Internet

Java™, C, C++, Visual Basic®, Object Technology, and Internet and World Wide Web Programming Training
On-Site Seminars Delivered Worldwide (Contact Us for on-site seminars on any programming topic)

deitel@deitel.com
978.579.9911
490-B Boston Post Road, Suite 200, Sudbury, MA 01776

Full-time Job Opportunities

Available Immediately for Computer Science and/or Information Technology Graduates at our Sudbury, MA location

Deitel & Associates, Inc. has several highly competitive **full-time positions** remaining.

Fig. 12.1 Sample Internet Explorer window with GUI components.

Control	Description
`Label`	An area in which icons or uneditable text can be displayed.
`TextBox`	An area in which the user inputs data from the keyboard. The area also can display information.
`Button`	An area that triggers an event when clicked.
`CheckBox`	A GUI control that is either selected or not selected.
`ComboBox`	A drop-down list of items from which the user can make a selection, by clicking an item in the list or by typing into the box, if permitted.
`ListBox`	An area in which a list of items is displayed from which the user can make a selection by clicking once on any element. Multiple elements can be selected.
`Panel`	A container in which components can be placed.
`ScrollBar`	Allows the user to access a range of values that cannot normally fit in its container.

Fig. 12.2 Some basic GUI components .

12.2 Windows Forms

Windows Forms (also called *WinForms*) create GUIs for programs. A form is a graphical element that appears on the desktop. A form can be a dialog, a window or an *MDI window* (*multiple document interface window*, discussed in Chapter 13, GUI Components: Part 2). A *component* is a class that implements the ***`IComponent`*** *interface*, which defines the behaviors that components must implement. A *control*, such as a button or label, is a component with a graphical part. Controls are visible, whereas components, which lack graphical parts, are not.

Figure 12.3 displays the Windows Forms controls and components contained in the Visual Studio .NET **Toolbox**—the first two screens show the controls and the last screen shows the components. When the user selects a component or control, the user then can add that component or control to the form. Note that the **Pointer** (the icon at the top of the list) is not a component; rather it represents the default mouse action. Highlighting it allows the programmer to use the mouse cursor instead of adding an item. In this chapter and the next, we discuss many of these controls.

When interacting with windows, we say that the *active window* has the *focus*. The active window is the frontmost window and has a highlighted title bar. A window becomes the active window when the user clicks somewhere inside it. When a window has focus, the operating system directs user input from the keyboard and mouse to that application.

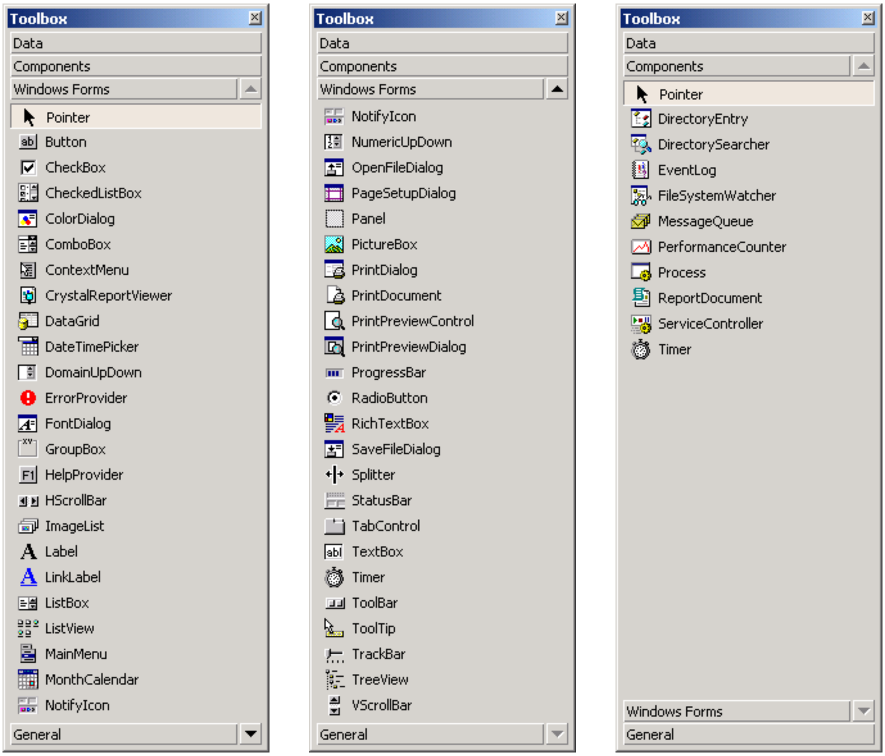

Toolbox
Data
Components
Windows Forms
Pointer
Button
CheckBox
CheckedListBox
ColorDialog
ComboBox
ContextMenu
CrystalReportViewer
DataGrid
DateTimePicker
DomainUpDown
ErrorProvider
FontDialog
GroupBox
HelpProvider
HScrollBar
ImageList
Label
LinkLabel
ListBox
ListView
MainMenu
MonthCalendar
NotifyIcon
General
Toolbox
Data
Components
Windows Forms
NotifyIcon
NumericUpDown
OpenFileDialog
PageSetupDialog
Panel
PictureBox
PrintDialog
PrintDocument
PrintPreviewControl
PrintPreviewDialog
ProgressBar
RadioButton
RichTextBox
SaveFileDialog
Splitter
StatusBar
TabControl
TextBox
Timer
ToolBar
ToolTip
TrackBar
TreeView
VScrollBar
General
Toolbox
Data
Components
Pointer
DirectoryEntry
DirectorySearcher
EventLog
FileSystemWatcher
MessageQueue
PerformanceCounter
Process
ReportDocument
ServiceController
Timer
Windows Forms
General

Fig. 12.3 Components and controls for Windows Forms.

The form acts as a *container* for components and controls. Controls must be added to the form using code. When we drag a control from the **Toolbox** onto the form, Visual Studio .NET generates this code for us, which instantiates the control and sets the control's basic properties. We could write the code ourselves, but it is much easier to create and modify controls using the **Toolbox** and **Properties** window, letting Visual Studio .NET handle the details. We introduced such *visual programming* earlier in the book. In the next several chapters, we build much richer GUIs through visual programming.

When the user interacts with a control by using the mouse or keyboard, events (discussed in Section 12.3) are generated, and event handlers process those events. Events typically cause something to happen in response. For example, clicking the **OK** button in a **MessageBox** generates an event. An event handler in class **MessageBox** closes the **MessageBox** in response to this event.

Each .NET Framework class (i.e., form, component and control) we present in this chapter is in the ***System.Windows.Forms*** *namespace*. Class **Form**, the basic window used by Windows applications, is fully qualified as **System.Windows.Forms.Form**. Likewise, class **Button** is actually **System.Windows.Forms.Button**.

The general design process for creating Windows applications requires creating a Windows Form, setting its properties, adding controls, setting their properties and implementing the event handlers. Figure 12.4 lists common **Form** properties and events.

Form Properties and Events	**Description / Delegate and Event Arguments**
Common Properties	
AcceptButton	Which button will be clicked when *Enter* is pressed.
AutoScroll	Whether scrollbars appear when needed (if data fill more than one screen).
CancelButton	Button that is clicked when the *Escape* key is pressed.
FormBorderStyle	Border of the form (e.g., **none**, **single**, **3D**, **sizable**).
Font	Font of text displayed on the form, as well as the default font of controls added to the form.
Text	Text in the form's title bar.
Common Methods	
Close	Closes form and releases all resources. A closed form cannot be reopened.
Hide	Hides form (does not release resources).
Show	Displays a hidden form.
Common Events	*(Delegate* ***EventHandler****, event arguments* ***EventArgs****)*
Load	Occurs before a form is shown. Visual Studio .NET generates a default event handler when the programmer double clicks on the form in the designer.

Fig. 12.4 Common **Form** properties and events.

Visual Studio .NET generates most GUI-related code when we create controls and event handlers. Programmers can use Visual Studio .NET to perform most of these tasks graphically, by dragging and dropping components onto the form and setting properties in the **Properties** window. In visual programming, the IDE generally maintains GUI-related code, and the programmer writes the event handlers.

12.3 Event-Handling Model

GUIs are *event driven* (i.e., they generate *events* when the program's user interacts with the GUI). Typical interactions include moving the mouse, clicking the mouse, clicking a button, typing in a textbox, selecting an item from a menu and closing a window. Event handlers are methods that process events and perform tasks. For example, consider a form that changes color when a button is clicked. When clicked, the button generates an event and passes it to the event handler, and the event-handler code changes the form's color.

Each control that can generate events has an associated delegate that defines the signature for that control's event handlers. Recall from Chapter 10 that delegates are objects that reference methods. Event delegates are *multicast* (class **`MulticastDelegate`**)—they contain lists of method references. Each method must have the same *signature* (i.e., the same list of parameters). In the event-handling model, delegates act as intermediaries between objects that generate events and methods that handle those events (Fig. 12.5).

Software Engineering Observation 12.1

Delegates enable classes to specify methods that will not be named or implemented until the class is instantiated. This is extremely helpful in creating event handlers. For instance, the creator of the **`Form`** *class does not need to name or define the method that will handle the* **`Click`** *event. Using delegates, the class can specify when such an event handler would be called. The programmers that create their own forms then can name and define this event handler. As long as it has been registered with the proper delegate, the method will be called at the proper time.*

Once an event is raised, every method that the delegate references is called. Every method in the delegate must have the same signature, because they are all passed the same information.

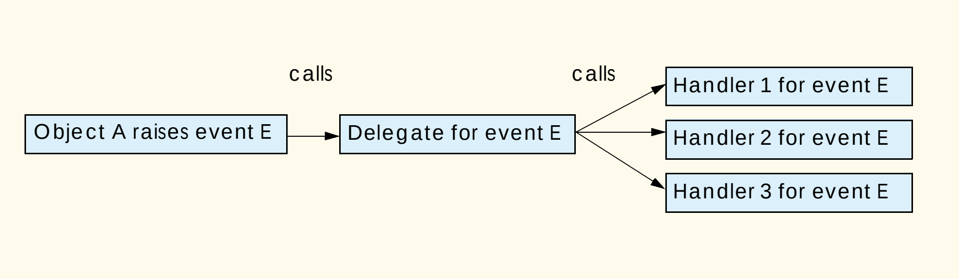


Fig. 12.5 Event-handling model using delegates.

12.3.1 Basic Event Handling

In most cases, we do not have to create our own events. Instead, we can handle the events generated by .NET controls such as buttons and text boxes. These controls already have delegates for every event they can raise. The programmer creates the event handler and registers it with the delegate—Visual Studio .NET helps with this task. In the following example, we create a form that displays a message box when clicked. Afterwards, we will analyze the event code that Visual Studio .NET generates.

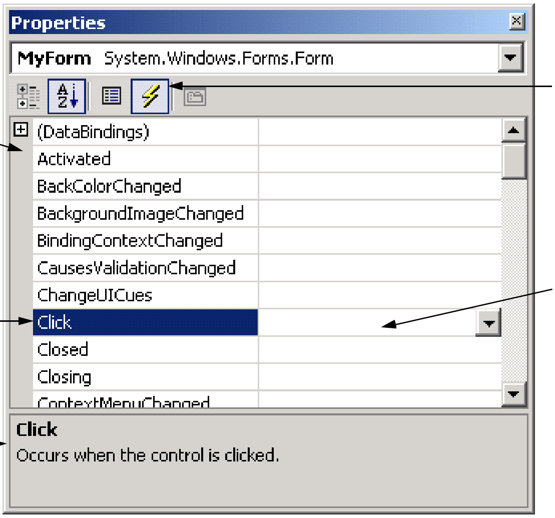
First, create a new Windows application. To register and define an event handler, click the **Events** icon (the yellow lightning bolt) in the form's **Properties** window (Fig. 12.6). This window allows the programmer to access, modify and create event handlers for a control. The left panel lists the events that the object can generate. The right panel lists the registered event handlers for the corresponding event; this list is initially empty. The drop-down button indicates that multiple handlers can be registered for one event. A brief description of the event appears on the bottom of the window.

In this example, the form will take some action when clicked. Double-click the **Click** event in the **Properties** window to create an empty event handler in the program code.

```
private void FormName_Click( object sender, System.EventArgs e )
{

}
```

This is the method that will be called when the form is clicked. As a response, we will have the form display a message box. To do this, insert the statement

```
MessageBox.Show( "Form was pressed." );
```


Properties
MyForm System.Windows.Forms.Form
(DataBindings)
Activated
BackColorChanged
BackgroundImageChanged
BindingContextChanged
CausesValidationChanged
ChangeUICues
Click
Closed
Closing
ContextMenuChanged

Click
Occurs when the control is clicked.

Events icon

List of events supported by control

Current event handler (none)

Selected event

Event description

Fig. 12.6 Events section of the **Properties** window.

into the event handler to get

```
private void FormName_Click( object sender, System.EventArgs e )
{
   MessageBox.Show( "Form was pressed" );
}
```

We can now compile and execute the program, which appears in Fig. 12.7. Whenever the form is clicked, a message box appears.

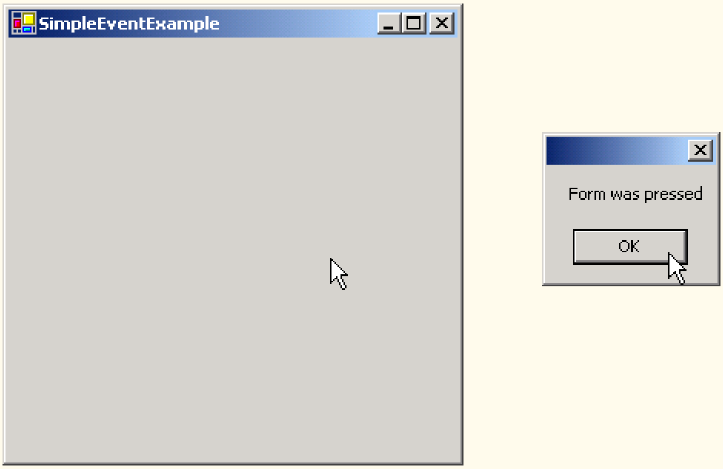
We now discuss the details of the program. First, we create an event handler (lines 26–29). Every event handler must have the signature that the corresponding event delegate specifies. Event handlers are passed two object references. The first is a reference to the object that raised the event (`sender`), and the second is a reference to an event arguments object (`e`). Argument `e` is of type `EventArgs`. Class `EventArgs` is the base class for objects that contain event information.

To create the event handler, we must find the delegate's signature. When we double-click an event name in the **Properties** window, Visual Studio .NET creates a method with the proper signature. The naming convention is *ControlName*_*EventName*; in our case the event handler is `MyForm_Click`. If we do not use the **Properties** window, we must look up the event arguments class. Consult the documentation index under *ControlName* **class** (i.e., **Form class**) and click the **events** section (Fig. 12.8). This displays a list of all the events the class can generate. Click the name of an event to bring up its delegate, event argument type and a description (Fig. 12.9).

```
 1  // Fig. 12.7: SimpleEventExample.cs
 2  // Using Visual Studio .NET to create event handlers.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  // program that shows a simple event handler
12  public class MyForm : System.Windows.Forms.Form
13  {
14     private System.ComponentModel.Container components = null;
15
16     // Visual Studio .NET generated code
17
18     [STAThread]
19     static void Main()
20     {
21        Application.Run( new MyForm() );
22     }
23
```

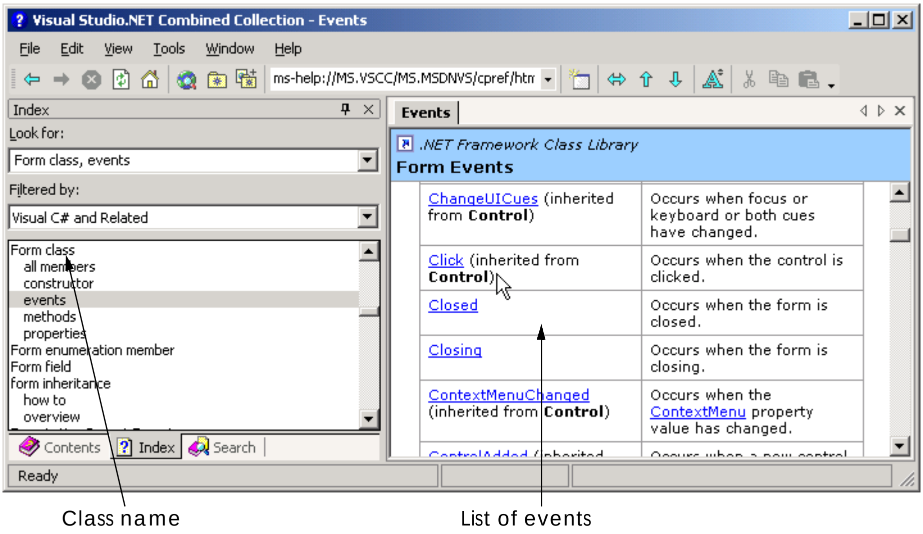
Fig. 12.7 Simple event-handling example using visual programming. (Part 1 of 2.)

```
      // Visual Studio .NET creates an empty handler,
      // we write definition: show message box when form clicked
      private void MyForm_Click( object sender, System.EventArgs e )
      {
         MessageBox.Show( "Form was pressed" );
      }

   } // end class MyForm
```


SimpleEventExample
OK

Form was pressed

Fig. 12.7 Simple event-handling example using visual programming. (Part 2 of 2.)


Visual Studio.NET Combined Collection - Events
File
Edit
View
Tools
Window
Help
ms-help://MS.VSCC/MS.MSDNVS/cpref/htm
Index
Form class
all members
constructor
events
methods
properties
Form enumeration member
Form field
form inheritance
how to
overview
Contents
Index
Search
Ready
Events
.NET Framework Class Library
Class name
List of events

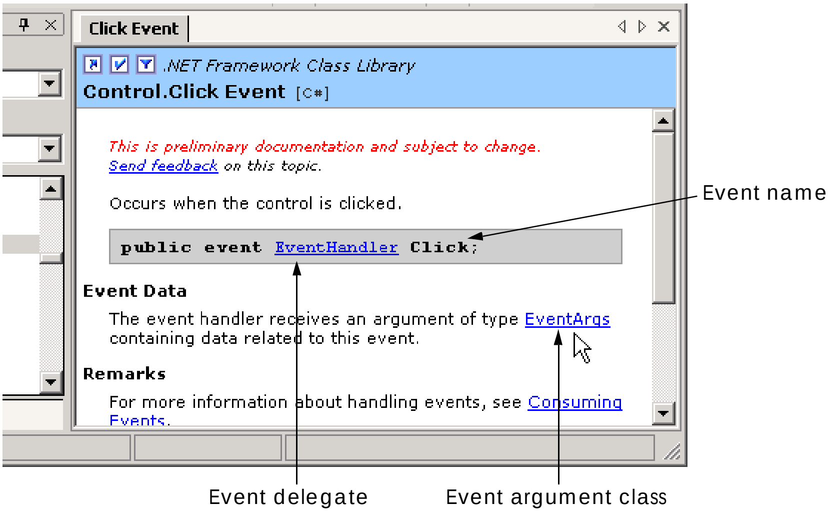
Look for: Form class, events

Filtered by: Visual C# and Related

Form Events

ChangeUICues (inherited from **Control**)	Occurs when focus or keyboard or both cues have changed.
Click (inherited from **Control**)	Occurs when the control is clicked.
Closed	Occurs when the form is closed.
Closing	Occurs when the form is closing.
ContextMenuChanged (inherited from **Control**)	Occurs when the ContextMenu property value has changed.

Fig. 12.8 List of **`Form`** events.



.NET Framework Class Library
Control.Click Event [C#]

This is preliminary documentation and subject to change.
Send feedback on this topic.

Occurs when the control is clicked.

```
public event EventHandler Click;
```

Event Data

The event handler receives an argument of type EventArgs containing data related to this event.

Remarks

For more information about handling events, see Consuming Events.

Fig. 12.9 Details of **`Click`** event.

The format of the event-handling method is, in general,

```
void ControlName_EventName( object sender, EventArgs e )
{
   event-handling code
}
```

where the name of the event handler is by default the name of the control, followed by an underscore (_) and the name of the event. Event handlers have return type **`void`** and take two arguments—an **`object`** (usually **`sender`**) and an instance of an event argument class. The differences between the various **`EventArgs`** classes are discussed in the following sections.

Good Programming Practice 12.1

Use the event-handler naming convention ControlName_EventName *to keep methods organized. This tells a user which event a method handles, and for which control. Visual Studio .NET uses this naming convention when creating event handlers from the* **Properties** *window.*

After creating the event handler, we must *register* it with the delegate object, which contains a list of event handlers to call. Registering an event handler with a delegate object involves adding the event handler to the delegate's invocation list. Controls have a *delegate reference* for each of their events—the delegate reference has the same name as the event. For example, if we are handling event *EventName* for object **`myControl`**, then the delegate reference is **`myControl.`***EventName*. Visual Studio .NET registers events for us with code such as the following from method **`InitializeComponent`**:

```
this.Click += new System.EventHandler( this.MyForm_Click );
```

The left-hand side is the delegate reference **`MyForm.Click`**. (**`this`** refers to an object of class **`MyForm`**.) The delegate reference is initially empty—we must assign to it an object reference (the right-hand side). We must create a new delegate object for each event handler. We create a new delegate object by writing

```
new System.EventHandler( methodName )
```

which returns a delegate object initialized with method *methodName*. The *methodName* is the name of the event handler, in our case it is **`MyForm.MyForm_Click`**. The **`+=`** operator adds an **`EventHandler`** delegate to the current delegate's invocation list. Since the delegate reference is initially empty, registering the first event handler creates a delegate object. In general, to register an event handler, write

```
objectName.EventName += new System.EventHandler(
   MyEventHandler );
```

We can add more event handlers using similar statements. *Event multicasting* is the ability to have multiple handlers for one event. Each event handler is called when the event occurs, but the order in which the event handlers are called is indeterminate. Use the **`-=`** operator to remove the method from the delegate object.

Common Programming Error 12.1

Assuming that multiple event handlers registered for the same event are called in a particular order can lead to logic errors. If the order is important, register the first event handler and have it call the others in order, passing the sender and event arguments.

Software Engineering Observation 12.2

*Events for prepackaged .NET components usually have consistent naming schemes. If the event is named **`EventName`**, then its delegate is **`EventNameEventHandler`**, and the event arguments class is **`EventNameEventArgs`**. However, events that use class **`EventArgs`** use delegate **`EventHandler`**.*

To review: The information needed to register an event is the **`EventArgs`** class (a parameter for the event handler) and the **`EventHandler`** delegate (to register the event handler). Visual Studio .NET can create this code for us, or we can type it in ourselves. If Visual Studio .NET creates the code, the programmer does not have to deal with going through all the steps, but the programmer also does not have complete control of everything that is going on. For simple events and event handlers it is often easier to allow Visual Studio .NET to generate this code. For more complicated solutions, registering your own event handlers might be necessary. In the upcoming sections, we will indicate the **`EventArgs`** class and the **`EventHandler`** delegate for each event we cover. To find more information about a particular type of event, search the help documentation for *ClassName* **class** and refer to the **events** subcategory.

12.4 Control Properties and Layout

This section overviews properties that are common to many controls. Controls derive from class ***`Control`*** (namespace **`System.Windows.Forms`**). Figure 12.10 contains a list of common properties and events for class **`Control`**. The **`Text`** property specifies the text that appears on a control, which may vary depending on the context. For example, the text of a Windows Form is its title bar, and the text of a button appears on its face. The ***`Focus`*** method transfers the focus to a control. When the focus is on a control, it becomes the active control. When the *Tab* key is pressed, the **`TabIndex`** property determines the order in which controls are given focus. The **`TabIndex`** property is automatically set by Visual Studio .NET, but can be changed by the programmer. This is helpful for the user who enters information in many different locations—the user can enter information and quickly select the next control by pressing the *Tab* key. The **`Enabled`** property indicates whether the control can be used. Programs can set property **`Enabled`** to false when an option is unavailable to the user. In most cases, the control's text will appear gray (rather than black), when a control is disabled. Without having to disable a control, the control can be hidden from the user by setting the **`Visible`** property to **`false`** or by calling method **`Hide`**. When a control's **`Visible`** property is set to **`false`**, the control still exists, but it is not shown on the form.

Class `Control` Properties and Methods	**Description**
Common Properties	
`BackColor`	Background color of the control.
`BackgroundImage`	Background image of the control.
`Enabled`	Whether the control is enabled (i.e., if the user can interact with it). A disabled control will still be displayed, but "grayed-out"—portions of the control will become gray.
`Focused`	Whether a control has focus. (The control that is currently being used in some way.)
`Font`	**`Font`** used to display control's **`Text`**.
`ForeColor`	Foreground color of the control. This is usually the color used to display the control's **`Text`** property.
`TabIndex`	Tab order of the control. When the *Tab* key is pressed, the focus is moved to controls in increasing tab order. This order can be set by the programmer.
`TabStop`	If **`true`**, user can use the *Tab* key to select the control.
`Text`	Text associated with the control. The location and appearance varies with the type of control.
`TextAlign`	The alignment of the text on the control. One of three horizontal positions (left, center or right) and one of three vertical positions (top, middle or bottom).
`Visible`	Whether the control is visible.
Common Methods	
`Focus`	Transfers the focus to the control.
`Hide`	Hides the control (sets **`Visible`** to **`false`**).
`Show`	Shows the control (sets **`Visible`** to **`true`**).

Fig. 12.10 Class **`Control`** properties and methods.

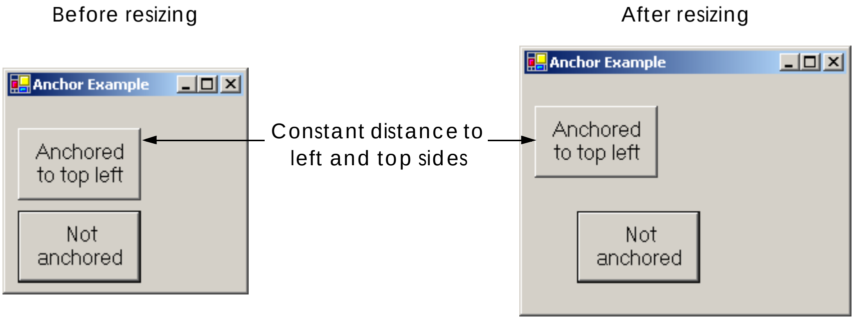
Visual Studio .NET allows the programmer to *anchor* and *dock* controls, which help to specify the layout of controls inside a container (such as a form). Anchoring allows controls to stay a fixed distance from the sides of the container, even when the control is resized. Docking allows controls to extend themselves along the sides of their containers.

A user may want a control to appear in a certain position (top, bottom, left or right) in a form even when that form is resized. The user can specify this by *anchoring* the control to a side (top, bottom, left or right). The control then maintains a fixed distance from the side to its parent container. In most cases, the parent container is a form; however, other controls can act as a parent container.

When parent containers are resized, all controls move. Unanchored controls move relative to their original position on the form, while anchored controls move so that they will be the same distance from each side that they are anchored to. For example, in Fig. 12.11, the topmost button is anchored to the top and left sides of the parent form. When the form is resized, the anchored button moves so that it remains a constant distance from the top and left sides of the form (its parent). The unanchored button changes position as the form is resized.

Create a simple Windows application that contains two controls. Anchor one control to the right side by setting the **Anchor** property as shown in Fig. 12.12. Leave the other control unanchored. Now, resize the form by dragging the right side farther to the right. Notice that both controls move. The anchored control moves so that it is always the same distance to the right wall. The unanchored control moves so that it is in the same place on the form, relative to each side. This control will continue to be somewhat closer to whatever sides it was originally close to, but will still reposition itselft when the user resizes the application window.

Sometimes a programmer wants a control to span the entire side of the form, even when the form is resized. This is useful when we want one control to remain prevalent on the form, such as the status bar that might appear at the bottom of a program. *Docking* allows a control to spread itself along an entire side (left, right, top or bottom) of its parent container. When the parent is resized, the docked control resizes as well. In Fig. 12.13, a button is docked to the top of the form. (It lays across the top portion.) When the form is resized, the button is resized as well—the button always fills the entire top portion of the form. The ***`Fill`*** dock option effectively docks the control to all sides of its parent, which causes it to fill its entire parent. Windows Forms contain property ***`DockPadding`***, which sets the distance from docked controls to the edge of the form. The default value is zero, causing the controls to attach to the edge of the form. The control layout properties are summarized in Fig. 12.14.

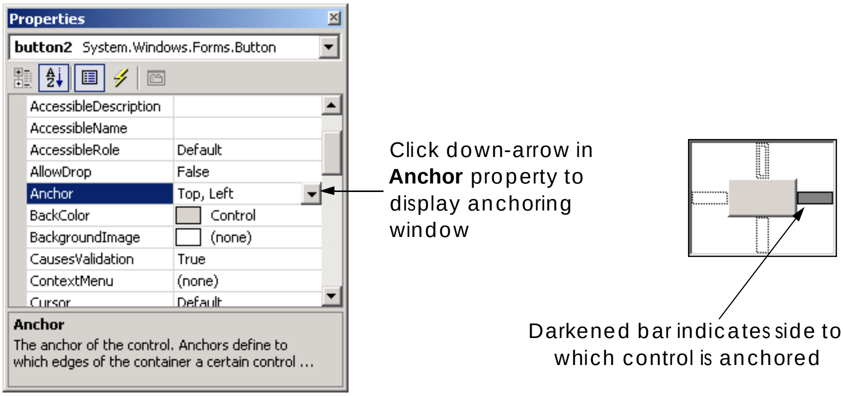


Before resizing

After resizing

Constant distance to left and top sides

Fig. 12.11 Anchoring demonstration.



Properties

button2 System.Windows.Forms.Button

AccessibleDescription	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
BackColor	Control
BackgroundImage	(none)
CausesValidation	True
ContextMenu	(none)
Cursor	Default

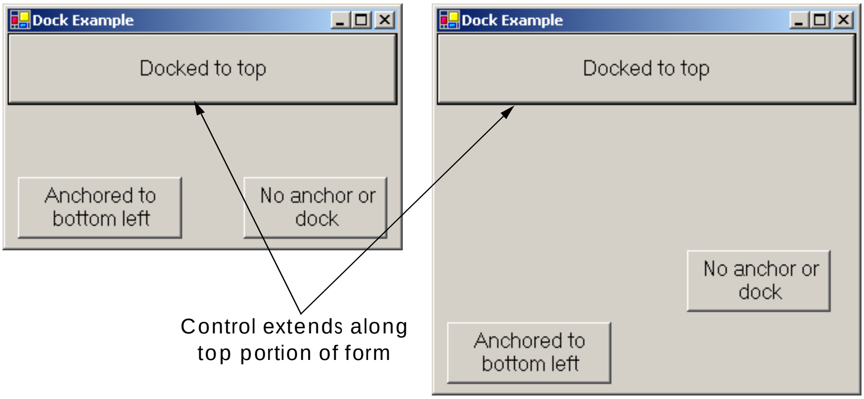
Anchor
The anchor of the control. Anchors define to which edges of the container a certain control ...

Click down-arrow in **Anchor** property to display anchoring window

Darkened bar indicates side to which control is anchored

Fig. 12.12 Manipulating the **Anchor** property of a control.

Before resizing After resizing



Dock Example
Docked to top
Anchored to bottom left
No anchor or dock
Dock Example
Docked to top
No anchor or dock
Anchored to bottom left

Control extends along top portion of form

Fig. 12.13 Docking demonstration.

Common Layout Properties	Description
Common Properties	
`Anchor`	Side of parent container at which to anchor control—values can be combined, such as `Top, Left`.
`Dock`	Side of parent container to dock control—values cannot be combined.
`DockPadding` (for containers)	Sets the dock spacing for controls inside the container. Default is zero, so controls appear flush against the side of the container.
`Location`	Location of the upper left corner of the control, relative to its container.

Fig. 12.14 Class `Control` layout properties. (Part 1 of 2.)

Common Layout Properties	**Description**
`Size`	Size of the control. Takes a **`Size`** structure, which has properties **`Height`** and **`Width`**.
`MinimumSize`, **`MaximumSize`** (for Windows Forms)	The minimum and maximum size of the form.

Fig. 12.14 Class **`Control`** layout properties. (Part 2 of 2.)

The docking and anchoring options refer to the parent container, which may or may not be the form. (We learn about other parent containers later this chapter.) The minimum and maximum form sizes can be set using properties **`MinimumSize`** and **`MaximumSize`**, respectively. Both properties use the **`Size`** structure, which has properties **`Height`** and **`Width`**, specifying the size of the form. These properties allow the programmer to design the GUI layout for a given size range. To set a form to a fixed size, set its minimum and maximum size to the same value.

Look-and-Feel Observation 12.2

Allow Windows forms to be resized—this enables users with limited screen space or multiple applications running at once to use the application more easily. Check that the GUI layout appears consistent for all permissible form sizes.

12.5 Labels, TextBoxes and Buttons

Labels provide text instructions or information about the program. Labels are defined with class ***`Label`***, which derives from class **`Control`**. A **`Label`** displays *read-only text*, or text that the user cannot modify. Once labels are created, programs rarely change their contents. Figure 12.15 lists common **`Label`** properties.

A *textbox* (class ***`TextBox`***) is an area in which text can be either input by the user from the keyboard or displayed. A *password textbox* is a **`TextBox`** that hides what the user entered. As the user types in characters, the password textbox displays only a certain character (usually **`*`**). Altering the ***`PasswordChar`*** property of a textbox makes it a password textbox and sets the appropriate character to be displayed. Deleting the value of **`PasswordChar`** in the **Properties** window sets the textbox back to a regular textbox. Figure 12.16 lists the common properties and events of **`TextBox`**es.

A *button* is a control that the user clicks to trigger a specific action. A program can use several other types of buttons, such as *checkboxes* and *radio buttons*. All the button types are derived from ***`ButtonBase`*** (namespace **`System.Windows.Forms`**), which defines common button features. In this section, we concentrate on the class **`Button`**, which is often used to initiate a command. The other button types are covered in subsequent sections. The text on the face of a **`Button`** is called a *button label*. Figure 12.17 lists the common properties and events of **`Button`**s.

`Label` Properties	**Description / Delegate and Event Arguments**
Common Properties	
`Font`	The font used by the text on the **`Label`**.
`Text`	The text to appear on the **`Label`**.
`TextAlign`	The alignment of the **`Label`**'s text on the control. One of three horizontal positions (**`left`**, **`center`** or **`right`**) and one of three vertical positions (**`top`**, **`middle`** or **`bottom`**).

Fig. 12.15 `Label` properties.

`TextBox` Properties and Events	**Description / Delegate and Event Arguments**
Common Properties	
`AcceptsReturn`	If **`true`**, pressing *Enter* creates a new line if textbox spans multiple lines. If **`false`**, pressing *Enter* clicks the default button of the form.
`Multiline`	If **`true`**, textbox can span multiple lines. Default is **`false`**.
`PasswordChar`	Single character to display instead of typed text, making the **`TextBox`** a password box. If no character is specified, **`Textbox`** displays the typed text.
`ReadOnly`	If **`true`**, **`TextBox`** has a gray background and its text cannot be edited. Default is **`false`**.
`ScrollBars`	For multiline textboxes, indicates which scrollbars appear (**`none`**, **`horizontal`**, **`vertical`** or **`both`**).
`Text`	The text to be displayed in the text box.
Common Events	*(Delegate **`EventHandler`**, event arguments **`EventArgs`**)*
`TextChanged`	Raised when text changes in **`TextBox`** (the user added or deleted characters). Default event when this control is double clicked in the designer.

Fig. 12.16 `TextBox` properties and events.

Look-and-Feel Observation 12.3

*Although **`Label`**s, **`TextBox`**es and other controls can respond to mouse-button clicks, **`Button`**s naturally convey this meaning. Use **`Button`**s (e.g., **OK**), rather than other types of controls, to initiate user actions.*

The program in Fig. 12.18 uses a **`TextBox`**, a **`Button`** and a **`Label`**. The user enters text into a password box and clicks the **`Button`**. The text then appears in the **`Label`**. Normally, we would not display this text—the purpose of password textboxes is to hide the text being entered by the user from anyone who may be looking over a person's shoulder.

`Button` properties and events	Description / Delegate and Event Arguments
Common Properties	
`Text`	Text displayed on the `Button` face.
Common Events	*(Delegate* ***`EventHandler`****, event arguments* ***`EventArgs`****)*
`Click`	Raised when user clicks the control. Default event when this control is double clicked in the designer.

Fig. 12.17 `Button` properties and events.

```
// Fig. 12.18: LabelTextBoxButtonTest.cs
// Using a Textbox, Label and Button to display
// the hidden text in a password box.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// namespace contains our form to display hidden text
namespace LabelTextBoxButtonTest
{
   /// <summary>
   /// form that creates a password textbox and
   /// a label to display textbox contents
   /// </summary>
   public class LabelTextBoxButtonTest :
      System.Windows.Forms.Form
   {
      private System.Windows.Forms.Button displayPasswordButton;
      private System.Windows.Forms.Label displayPasswordLabel;
      private System.Windows.Forms.TextBox inputPasswordTextBox;

      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container components = null;

      // default contructor
      public LabelTextBoxButtonTest()
      {
         InitializeComponent();
      }

```

Fig. 12.18 Program to display hidden text in a password box. (Part 1 of 4.)

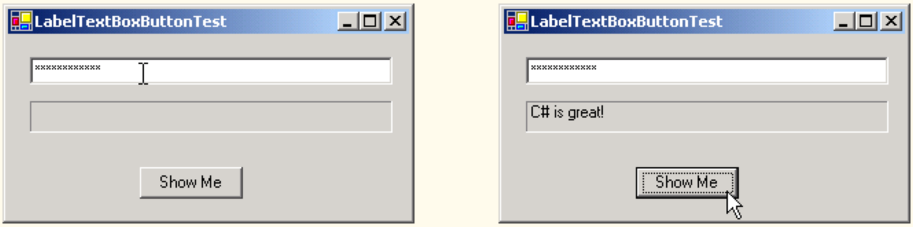
```
      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose( bool disposing )
      {
         if ( disposing )
         {
            if ( components != null )
            {
               components.Dispose();
            }
         }

         base.Dispose( disposing );
      }

      #region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.displayPasswordButton =
            new System.Windows.Forms.Button();
         this.inputPasswordTextBox =
            new System.Windows.Forms.TextBox();
         this.displayPasswordLabel =
            new System.Windows.Forms.Label();
         this.SuspendLayout();

         //
         // displayPasswordButton
         //
         this.displayPasswordButton.Location =
            new System.Drawing.Point( 96, 96 );
         this.displayPasswordButton.Name =
            "displayPasswordButton";
         this.displayPasswordButton.TabIndex = 1;
         this.displayPasswordButton.Text = "Show Me";
         this.displayPasswordButton.Click +=
            new System.EventHandler(
            this.displayPasswordButton_Click );

         //
         // inputPasswordTextBox
         //
         this.inputPasswordTextBox.Location =
            new System.Drawing.Point( 16, 16 );
         this.inputPasswordTextBox.Name =
            "inputPasswordTextBox";
         this.inputPasswordTextBox.PasswordChar = '*';
```

Fig. 12.18 Program to display hidden text in a password box. (Part 2 of 4.)

```
            this.inputPasswordTextBox.Size =
               new System.Drawing.Size( 264, 20 );
            this.inputPasswordTextBox.TabIndex = 0;
            this.inputPasswordTextBox.Text = "";

            //
            // displayPasswordLabel
            //
            this.displayPasswordLabel.BorderStyle =
               System.Windows.Forms.BorderStyle.Fixed3D;
            this.displayPasswordLabel.Location =
               new System.Drawing.Point( 16, 48 );
            this.displayPasswordLabel.Name =
               "displayPasswordLabel";
            this.displayPasswordLabel.Size =
               new System.Drawing.Size( 264, 23 );
            this.displayPasswordLabel.TabIndex = 2;

            //
            // LabelTextBoxButtonTest
            //
            this.AutoScaleBaseSize =
               new System.Drawing.Size( 5, 13 );
            this.ClientSize =
               new System.Drawing.Size( 292, 133 );
            this.Controls.AddRange(
               new System.Windows.Forms.Control[] {
                  this.displayPasswordLabel,
                  this.inputPasswordTextBox,
                  this.displayPasswordButton});
            this.Name = "LabelTextBoxButtonTest";
            this.Text = "LabelTextBoxButtonTest";
            this.ResumeLayout( false );

         } // end method InitializeComponent

         // end collapsible region started on line 53
         #endregion

         /// <summary>
         /// The main entry point for the application.
         /// </summary>
         [STAThread]
         static void Main()
         {
            Application.Run( new LabelTextBoxButtonTest() );
         }

         // display user input on label
         protected void displayPasswordButton_Click(
            object sender, System.EventArgs e )
         {
```

Fig. 12.18 Program to display hidden text in a password box. (Part 3 of 4.)

```
141             // text has not changed
142             displayPasswordLabel.Text =
143                inputPasswordTextBox.Text;
144          }
145
146       } // end class LabelTextBoxButtonTest
147
148    } // end namespace LabelTextBoxButtonTest
```



****

****

C# is great!

Fig. 12.18 Program to display hidden text in a password box. (Part 4 of 4.)

First, we create the GUI by dragging the components (a **`Button`**, a **`Label`** and a **`TextBox`**) onto the form. Once the components are positioned, we change their names in the **Properties** window (by setting the **`(Name)`** property) from the default values—**`textBox1`**, **`label1`**, **`button1`**—to the more descriptive **`displayPasswordLabel`**, **`inputPasswordTextBox`** and **`displayPasswordButton`**. Visual Studio .NET creates the code and places it inside method **`InitializeComponent`**. Now that the reader has an understanding of object-oriented programming, we can mention that the **`(Name)`** property is not really a property, but a means of changing the variable name of the object reference. For convenience, this value can be changed in the **Properties** window of Visual Studio .NET. This value, however, is not actually manipulated by a property.

We then set **`displayPasswordLabel`**'s **`Text`** property to "**`Show Me`**" and clear the **`Text`** of **`displayPasswordLabel`** and **`inputPasswordTextBox`** so that they are initially blank when the program runs. The **BorderStyle** property of **`displayPasswordLabel`** is set to **Fixed3D**, to give our **`Label`** a three-dimensional appearance. Notice that **`TextBox`**es have their **BorderStyle** property set to **Fixed3D** by default. The password character is set by assigning the asterisk character (**`*`**) to the **PasswordChar** property. This property can take only one character.

Let us examine the code that Visual Studio .NET generates by right-clicking the design and selecting **View Code**. This is important because not every change can be made in the **Properties** window.

We have learned in previous chapters that Visual Studio .NET adds comments to our code. These comments appear throughout the code, such as on lines 15–18. In future examples we remove some of these generated comments to make programs more concise and readable (unless they illustrate a capability we have not yet covered).

Visual Studio .NET inserts declarations for the controls we add to the form (lines 22–24), namely, the **`Label`**, **`TextBox`** and **`Button`**. The IDE manages these declarations for us, making it easy to add and remove controls. Line 29 declares reference **`components`**—an array to hold the components that we add. We are not using any components in this program (only controls), and thus the reference is **`null`**.

The constructor for our form is created for us—it calls method `InitializeComponent`. Method `InitializeComponent` creates the components and controls in the form and sets their properties. The usual "to do" comments generated by Visual Studio .NET have been removed, because there is no more code that needs to be added to the constructor. When they existed, they would have appeared as a reminder in the **Task List** window. Method `Dispose` cleans up allocated resources, but is not called explicitly in our programs.

Lines 53–126 contain a collapsible region that encloses our `InitializeComponent` method. Recall that the `#region` and `#endregion` preprocessor directives allow the programmer to collapse code to a single line in Visual Studio .NET. This enables the programmer to focus on certain portions of a program.

Method `InitializeComponent` (lines 58–123) sets the properties of the controls added to the form (the `TextBox`, `Label` and `Button`). Lines 60–66 create new objects for the controls we add (a `Button`, a `TextBox` and a `Label`). Lines 87–88 and 92 set the `Name`, `PasswordChar` and `Text` properties for `inputPasswordTextBox`. The `TabIndex` property is initially set by Visual Studio .NET, but can be changed by the developer.

The comment on lines 54–57 advises us not to modify the contents of method `InitializeComponent`. We have altered it slightly for formatting purposes in this book, but this is not recommended. We have done this only so that the reader is able to see the important portions of the code. Visual Studio .NET examines this method to create the design view of the code. If we change this method, Visual Studio .NET may not recognize our modifications and show the design improperly. It is important to note that the design view is based on the code, and not vice versa.

Testing and Debugging Tip 12.1

To keep the design view accurate, do not modify the code in method `InitializeComponent`. Make changes in the design window or property window.

The `Click` event is triggered when a control is clicked. We create the handler using the procedure described in Section 12.3.1. We want to respond to the `Click` event `displayPasswordButton`, so we double click it in the **Events** window. (Alternately, we could simply have clicked on `displayPasswordButton`.) This creates an empty event handler named `displayPasswordButton_Click` (line 138). Visual Studio .NET also registers the event handler for us (line 77–79). It adds the event handler to the `Click` event, using the `EventHandler` delegate. We must then implement the event handler. Whenever `displayPasswordButton` is clicked, this method is called and displays `inputPasswordTextBox`'s text on `displayPasswordLabel`. Even though `inputPasswordTextBox` displays all asterisks, it still retains its input text in its `Text` property. To show the text, we set `displayPasswordLabel`'s `Text` to `inputPasswordTextBox`'s `Text` (line 142–143). The user must program this line manually. When `displayPasswordButton` is clicked, the `Click` event is triggered, and the event handler `displayPasswordButton_Click` runs (updating `displayPasswordLabel`).

Visual Studio .NET generated most of the code in this program. It simplifies tasks such as creating controls, setting their properties and registering event handlers. However, we should be aware of how this is done—in several programs we may set properties ourselves, using code.

12.6 GroupBoxes and Panels

***GroupBox**es* and ***Panel**s* arrange components on a GUI. For example, buttons related to a particular task can be placed inside a **GroupBox** or **Panel** inside the Visual Studio .NET form designer. All these buttons move together when the **GroupBox** or **Panel** is moved.

The main difference between the two classes is that **GroupBox**es can display a caption, and **Panel**s can have scrollbars. The scrollbars allow the user to view additional controls inside the **Panel** by scrolling the visible area. **GroupBox**es have thin borders by default, but **Panel**s can be set to have borders by changing their **BorderStyle** property.

Look-and-Feel Observation 12.4

__Panel__s and __GroupBox__es can contain other __Panel__s and __GroupBox__es.

Look-and-Feel Observation 12.5

Organize the GUI by anchoring and docking controls (of similar function) inside a __GroupBox__ or __Panel__. The __GroupBox__ or __Panel__ then can be anchored or docked inside a form. This divides controls into functional "groups" that can be arranged easily.

To create a **GroupBox**, drag it from the toolbar and place it on a form. Create new controls and place them inside the **GroupBox**, causing them to become part of this class. These controls are added to the **GroupBox**'s **Controls** property. The **GroupBox**'s **Text** property determines its caption. The following tables list the common properties of **GroupBox**es (Fig. 12.19) and **Panel**s (Fig. 12.20).

GroupBox Properties	Description
Common Properties	
Controls	The controls that the **GroupBox** contains.
Text	Text displayed on the top portion of the **GroupBox** (its caption).

Fig. 12.19 **GroupBox** properties.

Panel Properties	Description
Common Properties	
AutoScroll	Whether scrollbars appear when the **Panel** is too small to hold its controls. Default is **false**.
BorderStyle	Border of the **Panel** (default **None**; other options are **Fixed3D** and **FixedSingle**).
Controls	The controls that the **Panel** contains.

Fig. 12.20 **Panel** properties.

To create a **`Panel`**, drag it onto the form and add components to it. To enable the scrollbars, set the **`Panel`**’s **`AutoScroll`** property to **`true`**. If the **`Panel`** is resized and cannot hold its controls, scrollbars appear (Fig. 12.21). These scrollbars then can be used to view all the components in the **`Panel`** (both when running and designing the form). This allows the programmer to see the GUI exactly as it appears to the client.

Look-and-Feel Observation 12.6

Use **`Panel`***s with scrollbars to avoid cluttering a GUI and to reduce the GUI’s size.*

The program in Fig. 12.22 uses a **`GroupBox`** and a **`Panel`** to arrange buttons. These buttons change the text on a **`Label`**.

The **`GroupBox`** (named **`mainGroupBox`**) has two buttons, **`hiButton`** (labeled **Hi**) and **`byeButton`** (labeled **Bye**). The **`Panel`** (named **`mainPanel`**) has two buttons as well, **`leftButton`** (labeled **Far Left**) and **`rightButton`** (labeled **Far Right**). The **`mainPanel`** control also has its **`AutoScroll`** property set to **True**, allowing scrollbars to appear if needed (i.e., if the contents of the **`Panel`** take up more space than the **`Panel`** itself). The **`Label`** (named **`messageLabel`**) is initially blank.

The event handlers for the four buttons are located in lines 36–61. To create an empty **`Click`** event handler, double click the button in design mode (instead of using the **Events** window). We add a line in each handler to change the text of **`messageLabel`**.

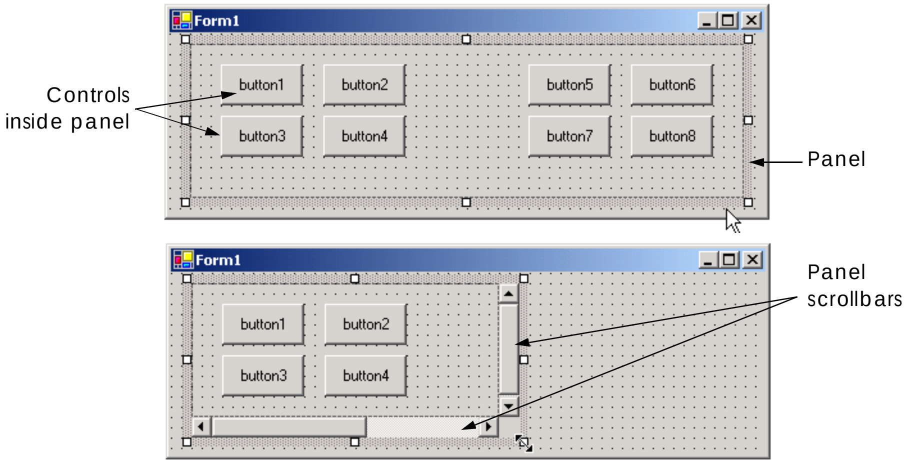


Fig. 12.21 Creating a **`Panel`** with scrollbars.

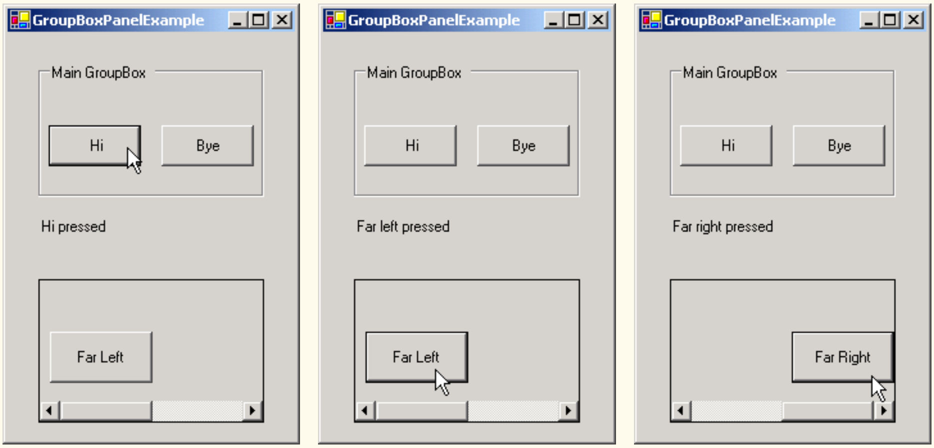
```
1  // Fig. 12.22: GroupBoxPanelExample.cs
2  // Using GroupBoxes and Panels to hold buttons.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
```

Fig. 12.22 Using **`GroupBox`**es and **`Panel`**s to arrange **`Button`**s. (Part 1 of 3.)

```
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

/// form to display a groupbox versus a panel
public class GroupBoxPanelExample : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button hiButton;
   private System.Windows.Forms.Button byeButton;
   private System.Windows.Forms.Button leftButton;
   private System.Windows.Forms.Button rightButton;

   private System.Windows.Forms.GroupBox mainGroupBox;
   private System.Windows.Forms.Label messageLabel;
   private System.Windows.Forms.Panel mainPanel;

   private System.ComponentModel.Container components = null;

   // Visual Studio .NET-generated Dispose method

   [STAThread]
   static void Main()
   {
      Application.Run( new GroupBoxPanelExample() );
   }

   // event handlers to change messageLabel

   // event handler for hi button
   private void hiButton_Click(
      object sender, System.EventArgs e )
   {
      messageLabel.Text= "Hi pressed";
   }

   // event handler for bye button
   private void byeButton_Click(
      object sender, System.EventArgs e )
   {
      messageLabel.Text = "Bye pressed";
   }

   // event handler for far left button
   private void leftButton_Click(
      object sender, System.EventArgs e )
   {
      messageLabel.Text = "Far left pressed";
   }

   // event handler for far right button
   private void rightButton_Click(
      object sender, System.EventArgs e )
   {
```

Fig. 12.22 Using `GroupBox`es and `Panel`s to arrange `Button`s. (Part 2 of 3.)

```
      messageLabel.Text = "Far right pressed";
   }

} // end class GroupBoxPanelExample
```


GroupBoxPanelExample
Main GroupBox
Hi
Bye
Far Left
GroupBoxPanelExample
Main GroupBox
Hi
Bye
Far Left
GroupBoxPanelExample
Main GroupBox
Hi
Bye
Far Right

Hi pressed

Far left pressed

Far right pressed

Fig. 12.22 Using **`GroupBox`**es and **`Panel`**s to arrange **`Button`**s. (Part 3 of 3.)

12.7 `CheckBox`es and `RadioButton`s

Visual C# has two types of *state buttons*—***`CheckBox`*** and ***`RadioButton`***—that can be in the on/off or true/false state. Classes **`CheckBox`** and **`RadioButton`** are derived from class **`ButtonBase`**. A **`RadioButton`** is different from a **`CheckBox`** in that there are normally several **`RadioButton`**s grouped together, and only one of the **`RadioButton`**s in the group can be selected (true) at any time.

A checkbox is a small white square that can be blank or contain a checkmark. When a checkbox is selected, a black checkmark appears in the box. There are no restrictions on how checkboxes are used: Any number may be selected at a time. The text that appears alongside a checkbox is referred to as the *checkbox label*. A list of common properties and events of class **`Checkbox`** appears in Fig. 12.23.

`CheckBox` events and properties	**Description / Delegate and Event Arguments**
Common Properties	
`Checked`	Whether the **`CheckBox`** has been checked.
`CheckState`	Whether the **`CheckBox`** is checked (contains a black checkmark) or unchecked (blank). An enumeration with values **`Checked`**, **`Unchecked`** or **`Indeterminate`**.
`Text`	Text displayed to the right of the **`CheckBox`** (called the label).

Fig. 12.23 **`CheckBox`** properties and events.

`CheckBox` events and properties	Description / Delegate and Event Arguments
Common Events	*(Delegate* ***`EventHandler`****, event arguments* ***`EventArgs`****)*
`CheckedChanged`	Raised every time the **`CheckBox`** is either checked or unchecked. Default event when this control is double clicked in the designer.
`CheckState-Changed`	Raised when the **`CheckState`** property changes.

Fig. 12.23 **`CheckBox`** properties and events.

The program in Fig. 12.24 allows the user to select a **`CheckBox`** to change the font style of a **`Label`**. One **`CheckBox`** applies a bold style, the other an italic style. If both checkboxes are selected, the style of the font is bold and italic. When the program initially executes, neither **`CheckBox`** is checked.

The first **`CheckBox`**, named **`boldCheckBox`**, has its **`Text`** property set to **`Bold`**. The other **`CheckBox`** is named **`italicCheckBox`** and is labeled **Italic**. The **`Label`**, named **`outputLabel`**, is labeled **Watch the font style change**.

```
// Fig. 12.24: CheckBoxTest.cs
// Using CheckBoxes to toggle italic and bold styles.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

/// form contains checkboxes to allow
/// the user to modify sample text
public class CheckBoxTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.CheckBox boldCheckBox;
   private System.Windows.Forms.CheckBox italicCheckBox;

   private System.Windows.Forms.Label outputLabel;

   private System.ComponentModel.Container components = null;

   // Visual Studio .NET-generated Dispose method

   /// The main entry point for the application.
   [STAThread]
   static void Main()
   {
      Application.Run( new CheckBoxTest() );
   }
```

Fig. 12.24 Using **`CheckBox`**es to change font styles. (Part 1 of 2.)

```
30
31       // make text bold if not bold,
32       // if already bold make not bold
33       private void boldCheckBox_CheckedChanged(
34          object sender, System.EventArgs e )
35       {
36          outputLabel.Font =
37             new Font( outputLabel.Font.Name,
38             outputLabel.Font.Size,
39             outputLabel.Font.Style ^ FontStyle.Bold );
40       }
41
42       // make text italic if not italic,
43       // if already italic make not italic
44       private void italicCheckBox_CheckedChanged(
45          object sender, System.EventArgs e )
46       {
47          outputLabel.Font =
48             new Font( outputLabel.Font.Name,
49             outputLabel.Font.Size,
50             outputLabel.Font.Style ^ FontStyle.Italic );
51       }
52
53    } // end class CheckBoxTest
```

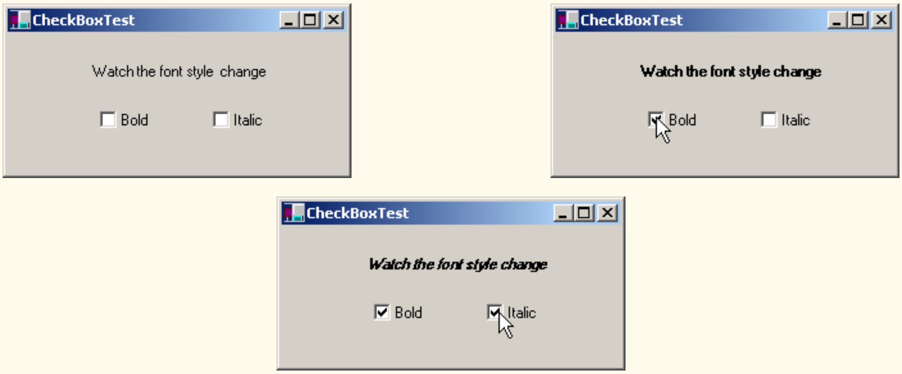

CheckBoxTest
Watch the font style change
☐ Bold
☐ Italic
CheckBoxTest
Watch the font style change
☑ Bold
☐ Italic
CheckBoxTest
Watch the font style change
☑ Bold
☑ Italic

Fig. 12.24 Using `CheckBox`es to change font styles. (Part 2 of 2.)

After creating the components, we define their event handlers. Double clicking **`boldCheckBox`** creates and registers an empty **`CheckedChanged`** event handler. To understand the code added to the event handler, we first discuss **`outputLabel`**'S **`Font`** property.

To change the font, the **`Font`** property must be set to a **`Font`** object. The **`Font`** constructor we use takes the font name, size and style. The first two arguments make use of **`outputLabel`**'s **`Font`** object, namely, **`outputLabel.Font.Name`** and **`outputLabel.Font.Size`** (lines 37–38). The style is a member of the **`FontStyle`** enumeration, which contains the font styles **`Regular`**, **`Bold`**, **`Italic`**, **`Strikeout`** and **`Underline`**. (The **`Strikeout`** style displays text with a line through it, the **`Underline`** style displays text with a line below it.) A **`Font`** object's **`Style`** property is set when the **`Font`** object is created—the **`Style`** property itself is read-only.

Styles can be combined using *bitwise operators*, or operators that perform manipulation on bits. Recall from Chapter 1 that all data are represented on the computer as a series of 0's and 1's. Each 0 or 1 is called a bit. Actions are taken and data are modified using these bit values. In this program, we need to set the font style so that the text will appear bold if it was not bold originally, and vice versa. Notice that on line 60 we use the bitwise XOR operator (^) to do this. Applying this operator to two bits does the following: If exactly 1 one of the corresponding bits is 1, set the result to 1. By using the ^ operator as we did on line 60, we are setting the bit values for bold in the same way. The operand on the right (`FontStyle.Bold`) always has bit values set to bold. The operand on the left, then (`outputLabel.Font.Style`) must not be bold for the resulting style to be bold. (Remember for XOR, if one value is set to 1, the other must be 0, or the result will not be 1.) If `outputLable.Font.Style` is bold, then the resulting style will not be bold. This operator also allows us to combine the styles. For instance, if the text were originally italicized, it would now be italicized and bold, rather than just bold.

We could have explicitly tested for the current style and changed it according to what we needed. For example, in the method `boldCheckBox_CheckChanged` we could have tested for the regular style, made it bold, tested for the bold style, made it regular, tested for the italic style, made it bold italic, or the italic bold style and made it italic. However, this method has a drawback—for every new style we add, we double the number of combinations. To add a checkbox for underline, we would have to test for eight possible styles. To add a checkbox for strikeout as well, we would have 16 tests in each event handler. By using the bitwise XOR operator, we save ourselves from this trouble. Each new style needs only a single statement in its event handler. In addition, styles can be removed easily, removing their handler. If we tested for every condition, we would have to remove the handler, and all the unnecessary test conditions in the other handlers.

Radio buttons (defined with class ***`RadioButton`***) are similar to checkboxes, because they also have two states—*selected* and *not selected* (also called *deselected*). However, radio buttons normally appear as a *group* in which only one radio button can be selected at a time. Selecting a different radio button in the group forces all other radio buttons in the group to be deselected. Radio buttons represent a set of *mutually exclusive* options (i.e., a set in which multiple options cannot be selected at the same time).

Look-and-Feel Observation 12.7

Use `RadioButton`s when the user should choose only one option in a group.

Look-and-Feel Observation 12.8

Use `CheckBox`es when the user should be able to choose many options in a group.

All radio buttons added to a form become part of the same group. To create new groups, radio buttons must be added to `GroupBox`es or `Panel`s. The common properties and events of class `RadioButton` are listed in Fig. 12.25.

Software Engineering Observation 12.3

Forms, `GroupBox`es, and `Panel`s can act as logical groups for radio buttons. The radio buttons within each group will be mutually exclusive to each other, but not to radio buttons in different groups.

`RadioButton` **properties and events**	**Description / Delegate and Event Arguments**
Common Properties	
`Checked`	Whether the **`RadioButton`** is checked.
`Text`	Text displayed to the right of the **`RadioButton`** (called the label).
Common Events	*(Delegate* ***`EventHandler`****, event arguments* ***`EventArgs`****)*
`Click`	Raised when user clicks the control.
`CheckedChanged`	Raised every time the **`RadioButton`** is checked or unchecked. Default event when this control is double clicked in the designer.

Fig. 12.25 **`RadioButton`** properties and events.

The program in Fig. 12.26 uses radio buttons to select the options for a **`MessageBox`**. Users select the attributes they want then press the display button, which causes the **`MessageBox`** to appear. A **`Label`** in the lower-left corner shows the result of the **`MessageBox`** (**Yes**, **No**, **Cancel**, etc.). The different **`MessageBox`** icon and button types have been displayed in tables in Chapter 5, Control Structures: Part 2.

```
// Fig. 12.26: RadioButtonsTest.cs
// Using RadioButtons to set message window options.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

/// form contains several radio buttons--user chooses one
/// from each group to create a custom MessageBox
public class RadioButtonsTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label promptLabel;
   private System.Windows.Forms.Label displayLabel;
   private System.Windows.Forms.Button displayButton;

   private System.Windows.Forms.RadioButton questionButton;
   private System.Windows.Forms.RadioButton informationButton;
   private System.Windows.Forms.RadioButton exclamationButton;
   private System.Windows.Forms.RadioButton errorButton;
   private System.Windows.Forms.RadioButton retryCancelButton;
   private System.Windows.Forms.RadioButton yesNoButton;
   private System.Windows.Forms.RadioButton yesNoCancelButton;
   private System.Windows.Forms.RadioButton okCancelButton;
   private System.Windows.Forms.RadioButton okButton;
```

Fig. 12.26 Using **`RadioButton`**s to set message-window options. (Part 1 of 5.)

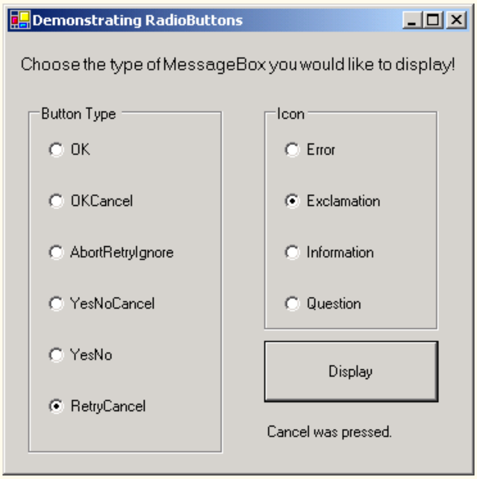
```
      private System.Windows.Forms.RadioButton
         abortRetryIgnoreButton;

      private System.Windows.Forms.GroupBox groupBox2;
      private System.Windows.Forms.GroupBox groupBox1;

      private MessageBoxIcon iconType = MessageBoxIcon.Error;
      private MessageBoxButtons buttonType =
         MessageBoxButtons.OK;

      /// The main entry point for the application.
      [STAThread]
      static void Main()
      {
         Application.Run( new RadioButtonsTest() );
      }

      // change button based on option chosen by sender
      private void buttonType_CheckedChanged(
         object sender, System.EventArgs e )
      {
         if ( sender == okButton ) // display OK button
            buttonType = MessageBoxButtons.OK;

         // display OK and Cancel buttons
         else if ( sender == okCancelButton )
            buttonType = MessageBoxButtons.OKCancel;

         // display Abort, Retry and Ignore buttons
         else if ( sender == abortRetryIgnoreButton )
            buttonType = MessageBoxButtons.AbortRetryIgnore;

         // display Yes, No and Cancel buttons
         else if ( sender == yesNoCancelButton )
            buttonType = MessageBoxButtons.YesNoCancel;

         // display Yes and No buttons
         else if ( sender == yesNoButton )
            buttonType = MessageBoxButtons.YesNo;

         // only one option left--display
         // Retry and Cancel buttons
         else
            buttonType = MessageBoxButtons.RetryCancel;

      } // end method buttonType_CheckedChanged

      // change icon based on option chosen by sender
      private void iconType_CheckedChanged(
         object sender, System.EventArgs e )
      {
         if ( sender == errorButton ) // display error icon
            iconType = MessageBoxIcon.Error;
```

Fig. 12.26 Using `RadioButton`s to set message-window options. (Part 2 of 5.)

```

         // display exclamation point
         else if ( sender == exclamationButton )
            iconType = MessageBoxIcon.Exclamation;

         // display information icon
         else if ( sender == informationButton )
            iconType = MessageBoxIcon.Information;

         else // only one option left--display question mark
            iconType = MessageBoxIcon.Question;

      } // end method iconType_CheckedChanged

      // display MessageBox and button user pressed
      protected void displayButton_Click(
         object sender, System.EventArgs e )
      {
         DialogResult result =
            MessageBox.Show( "This is Your Custom MessageBox.",
            "Custom MessageBox", buttonType, iconType, 0, 0 );

         // check for dialog result and display it in label
         switch ( result )
         {
            case DialogResult.OK:
               displayLabel.Text = "OK was pressed.";
               break;

            case DialogResult.Cancel:
               displayLabel.Text = "Cancel was pressed.";
               break;

            case DialogResult.Abort:
               displayLabel.Text = "Abort was pressed.";
               break;

            case DialogResult.Retry:
               displayLabel.Text = "Retry was pressed.";
               break;

            case DialogResult.Ignore:
               displayLabel.Text = "Ignore was pressed.";
               break;

            case DialogResult.Yes:
               displayLabel.Text = "Yes was pressed.";
               break;

            case DialogResult.No:
               displayLabel.Text = "No was pressed.";
               break;

```

Fig. 12.26 Using `RadioButton`s to set message-window options. (Part 3 of 5.)

```
         } // end switch

      } // end method displayButton_Click

   } // end class RadioButtonsTest
```



Demonstrating RadioButtons

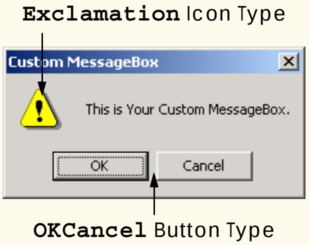
Choose the type of MessageBox you would like to display!

Button Type
○ OK
○ OKCancel
○ AbortRetryIgnore
○ YesNoCancel
○ YesNo
● RetryCancel

Icon
○ Error
● Exclamation
○ Information
○ Question

Display

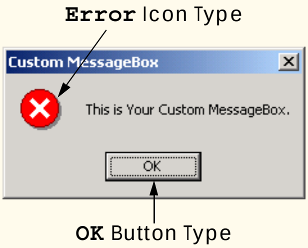
Cancel was pressed.



Custom MessageBox

This is Your Custom MessageBox.

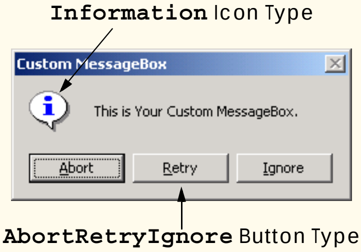
Exclamation Icon Type
OK
Cancel
OKCancel Button Type



Custom MessageBox

This is Your Custom MessageBox.

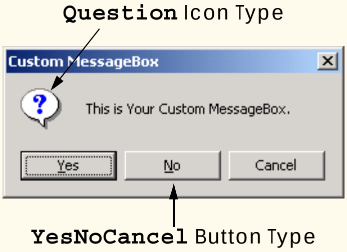
Error Icon Type
OK
OK Button Type



Custom MessageBox

This is Your Custom MessageBox.

Information Icon Type
Abort
Retry
Ignore
AbortRetryIgnore Button Type

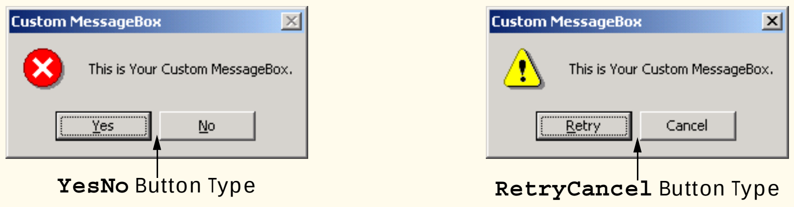


Custom MessageBox

This is Your Custom MessageBox.

Question Icon Type
Yes
No
Cancel
YesNoCancel Button Type

Fig. 12.26 Using `RadioButton`s to set message-window options. (Part 4 of 5.)


Custom MessageBox

This is Your Custom MessageBox.

Custom MessageBox

This is Your Custom MessageBox.

Fig. 12.26 Using `RadioButton`s to set message-window options. (Part 5 of 5.)

To store the user's choice of options, the objects `iconType` and `buttonType` are created and initialized (lines 34–36). Object `iconType` is a `MessageBoxIcon` enumeration that can have values `Asterisk`, `Error`, `Exclamation`, `Hand`, `Information`, `Question`, `Stop` and `Warning`. In this example we use only `Error`, `Exclamation`, `Information` and `Question`.

Object `buttonType` is a `MessageBoxButton` enumeration with values `AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, `YesNo` and `YesNoCancel`. The name indicates which buttons will appear in the `MessageBox`. In this example we use all `MessageBoxButton` enumeration values.

Two `GroupBox`es are created, one for each enumeration. Their captions are **Button Type** and **Icon**. One label is the used to prompt the user (`promptLabel`), while the other is used to display which button was pressed, once the custom `MessageBox` has been displayed (`displayLabel`). There is also a button (`displayButton`) that displays the text **Display**. `RadioButton`s are created for the enumeration options, with their labels set appropriately. The radio buttons are grouped, thus only one option can be selected from each `GroupBox`.

For event handling, one event handler exists for all the radio buttons in `groupBox1`, and another for all the radio buttons in `groupBox2`. Each radio button generates a `CheckedChanged` event when clicked.

Remember, to set the event handler for an event, use the events section of the **Properties** window. Create a new `CheckedChanged` event handler for one of the radio buttons in `buttonTypeGroupBox` and rename it `buttonType_CheckedChanged`. Then set the `CheckedChanged` event handlers for all the radio buttons in `buttonTypeGroupBox` to method `buttonType_CheckedChanged`. Create a second `CheckedChanged` event handler for a radio button in `iconTypeGroupBox` and rename it `iconType_CheckedChanged`. Finally, set the `CheckedChanged` event handlers for the radio buttons in `iconTypeGroupBox` to method `iconType_CheckedChanged`.

Both handlers compare the `sender` object with every radio button to determine which button was selected. Depending on the radio button selected, either `iconType` or `buttonType` changes (lines 46–93).

The `Click` handler for `displayButton` (lines 96–136) creates a `MessageBox` (lines 99–101). Some of the `MessageBox` options are set by `iconType` and `buttonType`. The result of the message box is a `DialogResult` enumeration, with values `Abort`, `Cancel`, `Ignore`, `No`, `None`, `OK`, `Retry` or `Yes`. The `switch` statement on lines 104–134 tests for the result and sets `displayLabel.Text` appropriately.

12.8 `PictureBox`es

A picture box (class **`PictureBox`**) displays an image. The image, set by an object of class **`Image`**, can be in a bitmap (**`.bmp`**), **`.gif`**, **`.jpg`**, icon or metafile format. (Images and multimedia are discussed in Chapter 16, Graphics and Multimedia.) *GIF (Graphics Interchange Format)* and *JPEG (Joint Photographic Expert Group)* files are widely used file formats.

The **`Image`** property sets the **`Image`** object to use, and the **`SizeMode`** property sets how the image is displayed (**`Normal`**, **`StretchImage`**, **`AutoSize`** or **`CenterImage`**). Figure 12.27 describes the important properties and events of class **`PictureBox`**.

The program in Fig. 12.28 uses **`PictureBox imagePictureBox`** to display one of three bitmap images—**`image0`**, **`image1`** or **`image2`**. They are located in the directory **`images`** (as usual, located in the **`bin/debug`** directory of our project), where the executable file is located. Whenever the **`imagePictureBox`** is clicked, the image changes. The **`Label`** (named **`promptLabel`**) on the top of the form includes the instructions **Click On Picture Box to View Images**.

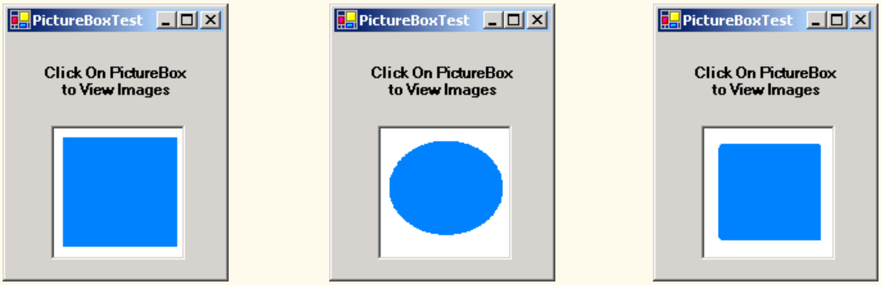
`PictureBox` **properties and events**	**Description / Delegate and Event Arguments**
Common Properties	
`Image`	Image to display in the **`PictureBox`**.
`SizeMode`	Enumeration that controls image sizing and positioning. Values **`Normal`** (default), **`StretchImage`**, **`AutoSize`** and **`CenterImage`**. **`Normal`** puts image in top-left corner of **`PictureBox`** and **`CenterImage`** puts image in middle. (Both cut off image if too large.) **`StretchImage`** resizes image to fit in **`PictureBox`**. **`AutoSize`** resizes **`PictureBox`** to hold image.
Common Events	*(Delegate **`EventHandler`**, event arguments **`EventArgs`**)*
`Click`	Raised when user clicks the control. Default event when this control is double clicked in the designer.

Fig. 12.27 `PictureBox` properties and events.

```
// Fig. 12.28: PictureBoxTest.cs
// Using a PictureBox to display images.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

```

Fig. 12.28 Using a `PictureBox` to display images. (Part 1 of 2.)

```
12  /// form to display different images when clicked
13  public class PictureBoxTest : System.Windows.Forms.Form
14  {
15     private System.Windows.Forms.PictureBox imagePictureBox;
16     private System.Windows.Forms.Label promptLabel;
17  
18     private int imageNum = -1;
19  
20     /// The main entry point for the application.
21     [STAThread]
22     static void Main()
23     {
24        Application.Run( new PictureBoxTest() );
25     }
26  
27     // change image whenever PictureBox clicked
28     private void imagePictureBox_Click(
29        object sender, System.EventArgs e )
30     {
31        imageNum = ( imageNum + 1 ) % 3; // imageNum from 0 to 2
32  
33        // create Image object from file, display on PictureBox
34        imagePictureBox.Image = Image.FromFile(
35           Directory.GetCurrentDirectory() + "\\images\\image" +
36           imageNum + ".bmp" );
37     }
38  
39  } // end class PictureBoxTest
```



Click On PictureBox to View Images

Fig. 12.28 Using a **`PictureBox`** to display images. (Part 2 of 2.)

To respond to the user's clicks, we must handle the **`Click`** event (lines 28–37). Inside the event handler, we use an integer (**`imageNum`**) to store the image we want to display. We then set the **`Image`** property of **`imagePictureBox`** to an **`Image`**. Class **`Image`** is discussed in Chapter 16, Graphics and Multimedia, but here we overview method **`FromFile`**, which takes a **`string`** (the path to the image file) and creates an **`Image`** object.

To find the images, we use class **`Directory`** (namespace **`System.IO`**, specified on line 10) method **`GetCurrentDirectory`** (line 35). This returns the current directory of the executable file (usually **`bin\Debug`**) as a **`string`**. To access the **`images`** subdirectory, we take the current directory and append "**`\\images`**" followed by "**`\\`**" and the file name. We use a double slash because an escape sequence is needed to print a single slash.

Alternatively, we could have used `@` to avoid the escape character (i.e., `@"\"` will print a single slash—the slash does not need to be escaped by another slash). We use `imageNum` to append the proper number, so we can load either `image0`, `image1` or `image2`. Integer `imageNum` stays between `0` and `2`, due to the modulus calculation (line 31). Finally, we append `".bmp"` to the filename. Thus, if we want to load `image0`, the string becomes "*CurrentDir*`\images\image0.bmp`", where *`CurrentDir`* is the directory of the executable.

12.9 Mouse Event Handling

This section explains how to handle *mouse events,* such as *clicks*, *presses* and *moves*. Mouse events are generated when the mouse interacts with a control. They can be handled for any GUI control that derives from class `System.Windows.Forms.Control`. Mouse event information is passed using class *`MouseEventArgs`*, and the delegate to create the mouse event handlers is *`MouseEventHandler`*. Each mouse event-handling method must take an `object` and a `MouseEventArgs` object as arguments. The `Click` event, which we covered earlier, uses delegate `EventHandler` and event arguments `EventArgs`.

Class `MouseEventArgs` contains information about the mouse event, such as the *x*- and *y*-coordinates of the mouse pointer, the mouse button pressed, the number of clicks and the number of notches through which the mouse wheel turned. Note that the *x*- and *y*-coordinates of the `MouseEventArgs` object are relative to the control that raised the event. Point (0,0) is at the upper-left corner of the control. The various mouse events are described in Fig. 12.29.

Mouse Events, Delegates and Event Arguments	
Mouse Events (Delegate `EventHandler`, event arguments `EventArgs`)	
`MouseEnter`	Raised if the mouse cursor enters the area of the control.
`MouseLeave`	Raised if the mouse cursor leaves the area of the control.
Mouse Events (Delegate `MouseEventHandler`, event arguments `MouseEventArgs`)	
`MouseDown`	Raised if the mouse button is pressed while its cursor is over the area of the control.
`MouseHover`	Raised if the mouse cursor hovers over the area of the control.
`MouseMove`	Raised if the mouse cursor is moved while in the area of the control.
`MouseUp`	Raised if the mouse button is released when the cursor is over the area of the control.
Class `MouseEventArgs` Properties	
`Button`	Mouse button that was pressed (`left`, `right`, `middle` or `none`).
`Clicks`	The number of times the mouse button was clicked.
`X`	The *x*-coordinate of the event, relative to the control.
`Y`	The *y*-coordinate of the event, relative to the control.

Fig. 12.29 Mouse events, delegates and event arguments.

Figure 12.30 uses mouse events to draw on the form. Whenever the user drags the mouse (i.e., moves the mouse while holding down a button), a line is drawn on the form.

```
// Fig 12.30: Painter.cs
// Using the mouse to draw on a form.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

/// creates a form as a drawing surface
public class Painter : System.Windows.Forms.Form
{
   bool shouldPaint = false; // whether to paint

   /// The main entry point for the application.
   [STAThread]
   static void Main()
   {
      Application.Run( new Painter() );
   }

   // should paint after mouse button has been pressed
   private void Painter_MouseDown(
      object sender, System.Windows.Forms.MouseEventArgs e )
   {
      shouldPaint = true;
   }

   // stop painting when mouse button released
   private void Painter_MouseUp(
      object sender, System.Windows.Forms.MouseEventArgs e )
   {
      shouldPaint = false;
   }

   // draw circle whenever mouse button
   // moves (and mouse is down)
   protected void Painter_MouseMove(
      object sender, System.Windows.Forms.MouseEventArgs e )
   {
      if ( shouldPaint )
      {
         Graphics graphics = CreateGraphics();
         graphics.FillEllipse(
            new SolidBrush( Color.BlueViolet ),
            e.X, e.Y, 4, 4 );
      }

   } // end Painter_MouseMove
```

Fig. 12.30 Using the mouse to draw on a form. (Part 1 of 2.)

```
51
52   } // end class Painter
```

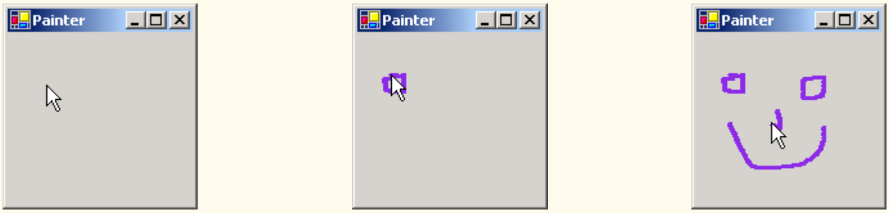


Fig. 12.30 Using the mouse to draw on a form. (Part 2 of 2.)

On line 14 the program creates variable **`shouldPaint`**, which determines whether we should draw on the form. We want to draw only while the mouse button is pressed down. In the event handler for event **`MouseDown`**, **`shouldPaint`** is set to true (line 27). As soon as the mouse button is released the program stops drawing: **`shouldPaint`** is set to false in the **`MouseUp`** event handler (line 34).

Whenever the mouse moves while the button is pressed down, the **`MouseMove`** event is generated. The event will be generated repeatedly, at a rate set by the operating system. Inside the **`Painter_MouseMove`** event handler (lines 39–48), the program draws only if **`shouldPaint`** is **`true`** (indicating that the mouse button is down). Line 44 creates the **`Graphics`** object for the form, which provides methods for drawing various shapes. Method **`FillEllipse`** (lines 45–47) draws a circle at every point the mouse cursor moves over (while the mouse button is pressed). The first parameter to method **`FillEllipse`** is a **`SolidBrush`** object, which determines the color of the shape drawn. We create a new **`SolidBrush`** object by passing the constructor a **`Color`** value. Structure **`Color`** contains numerous predefined color constants—we selected **`Color.BlueViolet`** (line 46). The **`SolidBrush`** fills an elliptical region, which lies inside a bounding rectangle. The bounding rectangle is specified by the *x*- and *y*-coordinates of its upper-left corner, its height and its width. These four parameters are the final four arguments to method **`FillEllipse`**. The *x*- and *y*-coordinates are the location of the mouse event: They can be taken from the mouse event arguments (**`e.X`** and **`e.Y`**). To draw a circle, we set the height and width of the bounding rectangle equal—in this case, they are each 4 pixels.

12.10 Keyboard Event Handling

This section explains how to handle *key events*. Key events are generated when keys on the keyboard are pressed and released. These events can be handled by any control that inherits from **`System.Windows.Forms.Control`**. There are two types of key events. The first is event ***`KeyPress`***, which fires when a key representing an ASCII character is pressed (determined by ***`KeyPressEventArgs`*** property ***`KeyChar`***). ASCII is a 128-character set of alphanumeric symbols. (The full listing can be found in Appendix B, ASCII Character Set.)

Using the **`KeyPress`** event, we cannot determine if *modifier keys* (such as *Shift*, *Alt* and *Control*) were pressed. To determine such actions, handle the ***`KeyUp`*** or ***`KeyDown`*** events, which form the second type of key event. Class ***`KeyEventArgs`*** contains information about special modifier keys. The key's ***`Key`*** *enumeration* value can be returned, giving information about a wide range of non-ASCII keys. Modifier keys are often used in conjunction with the mouse to select or highlight information. The delegates for the two classes are ***KeyPressEventHandler*** (event argument class **KeyPressEventArgs**) and ***KeyEventHandler*** (event argument class **KeyEventArgs**). Figure 12.31 lists important information about key events.

Figure 12.32 demonstrates using the key event handlers to display the key that was pressed. The program's form contains two **Label**s. It displays the key pressed on one **Label** and modifier information on the other.

The two **Label**s (named **charLabel** and **keyInfoLabel**) are initially empty. The **KeyDown** and **KeyPress** events convey different information; thus, the form (**KeyDemo**) handles them both.

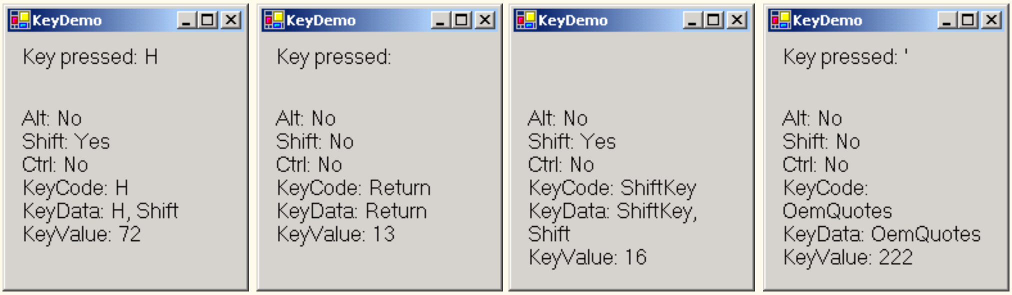
Keyboard Events, Delegates and Event Arguments	
*Key Events (Delegate **KeyEventHandler**, event arguments **KeyEventArgs**)*	
KeyDown	Raised when key is initially pushed down.
KeyUp	Raised when key is released.
*Key Events (Delegate **KeyPressEventHandler**, event arguments **KeyPressEventArgs**)*	
KeyPress	Raised when key is pressed. Occurs repeatedly while key is held down, at a rate specified by the operating system.
*Class **KeyPressEventArgs** Properties*	
KeyChar	Returns the ASCII character for the key pressed.
Handled	Whether the **KeyPress** event was handled.
*Class **KeyEventArgs** Properties*	
Alt	Indicates whether the *Alt* key was pressed.
Control	Indicates whether the *Control* key was pressed.
Shift	Indicates whether the *Shift* key was pressed.
Handled	Whether the event was handled.
KeyCode	Returns the key code for the key, as a **Keys** enumeration. This does not include modifier key information. Used to test for a specific key.
KeyData	Returns the key code as a **Keys** enumeration, combined with modifier information. Used to determine all information about the key pressed.
KeyValue	Returns the key code as an **int**, rather than as a **Keys** enumeration. Used to obtain a numeric representation of the key pressed.
Modifiers	Returns a **Keys** enumeration for any modifier keys pressed (*Alt*, *Control* and *Shift*). Used to determine modifier key information only.

Fig. 12.31 Keyboard events, delegates and event arguments.

```
// Fig. 12.32: KeyDemo.cs
// Displaying information about the key the user pressed.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// form to display key press
// information--contains two labels
public class KeyDemo : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label charLabel;
   private System.Windows.Forms.Label keyInfoLabel;

   private System.ComponentModel.Container components = null;

   /// The main entry point for the application.
   [STAThread]
   static void Main()
   {
      Application.Run( new KeyDemo() );
   }

   // display the character pressed using key char
   protected void KeyDemo_KeyPress(
      object sender, System.Windows.Forms.KeyPressEventArgs e )
   {
      charLabel.Text = "Key pressed: " + e.KeyChar;
   }

   // display modifier keys, key code, key data and key value
   private void KeyDemo_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      keyInfoLabel.Text =
         "Alt: " + ( e.Alt ? "Yes" : "No") + '\n' +
         "Shift: " + ( e.Shift ? "Yes" : "No" ) + '\n' +
         "Ctrl: " + ( e.Control ? "Yes" : "No" ) + '\n' +
         "KeyCode: " + e.KeyCode + '\n' +
         "KeyData: " + e.KeyData + '\n' +
         "KeyValue: " + e.KeyValue;
   }

   // clear labels when key released
   private void KeyDemo_KeyUp(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      keyInfoLabel.Text = "";
      charLabel.Text = "";
   }
```

Fig. 12.32 Demonstrating keyboard events (Part 1 of 2.).

```
54
55   } // end class KeyDemo
```



```
Key pressed: H

Alt: No
Shift: Yes
Ctrl: No
KeyCode: H
KeyData: H, Shift
KeyValue: 72
```

```
Key pressed:

Alt: No
Shift: No
Ctrl: No
KeyCode: Return
KeyData: Return
KeyValue: 13
```

```
Alt: No
Shift: Yes
Ctrl: No
KeyCode: ShiftKey
KeyData: ShiftKey,
Shift
KeyValue: 16
```

```
Key pressed: '

Alt: No
Shift: No
Ctrl: No
KeyCode:
OemQuotes
KeyData: OemQuotes
KeyValue: 222
```

Fig. 12.32 Demonstrating keyboard events (Part 2 of 2.).

The **`KeyPress`** event handler (lines 28–32) accesses the **`KeyChar`** property of the **`KeyPressEventArgs`** object. This returns the key pressed as a **`char`** and displays in **`charLabel`** (line 31). If the key pressed was not an ASCII character, then the **`KeyPress`** event will not fire and **`charLabel`** remains empty. ASCII is a common encoding format for letters, numbers, punctuation marks and other characters. It does not support keys such as the *function keys* (like *F1*) or the modifier keys (*Alt*, *Control* and *Shift*).

The **`KeyDown`** event handler (lines 35–45) displays more information, all from its **`KeyEventArgs`** object. It tests for the *Alt*, *Shift* and *Control* keys (lines 39–41), using the **`Alt`**, **`Shift`** and **`Control`** properties, each of which returns **`bool`**. It then displays the **`KeyCode`**, **`KeyData`** and **`KeyValue`** properties.

The **`KeyCode`** property returns a **`Keys`** enumeration, which is converted to a **`string`** using method **`ToString`**. The **`KeyCode`** property returns the key that was pressed, but does not provide any information about modifier keys. Thus, both a capital and a lowercase “a” are represented as the *A* key.

The **`KeyData`** property returns a **`Keys`** enumeration as well, but includes data about modifier keys. Thus, if “A” is input, the **`KeyData`** shows that the *A* key and the *Shift* key were pressed. Lastly, **`KeyValue`** returns the key code for the key that was pressed as an integer. This integer is the *Windows virtual key code*, which provides an integer value for a wide range of keys and for mouse buttons. The Windows virtual key code is useful when testing for non-ASCII keys (such as *F12*).

The **`KeyUp`** event handler clears both labels when the key is released (lines 48–53). As we can see from the output, non-ASCII keys are not displayed in the upper **`charLabel`** because the **`KeyPress`** event was not generated. The **`KeyDown`** event is still raised, and **`keyInfoLabel`** displays information about the key. The **`Keys`** enumeration can be used to test for specific keys by comparing the key pressed to a specific **`KeyCode`**. The Visual Studio. NET documentation has a complete list of the **`Keys`** enumerations.

Software Engineering Observation 12.4

To cause a control to react when a certain key is pressed (such as Enter*), handle a key event and test for the key pressed. To cause a button to be clicked when the* Enter *key is pressed on a form, set the form’s* ***`AcceptButton`*** *property.*

SUMMARY

- A graphical user interface (GUI) presents a pictorial interface to a program. A GUI (pronounced “GOO-EE”) gives a program a distinctive “look” and “feel.”
- By providing different applications with a consistent set of intuitive user interface components, GUIs allow the user to concentrate on using programs productively.
- GUIs are built from GUI components (sometimes called controls or widgets). A GUI control is a visual object with which the user interacts via the mouse or keyboard.
- Windows Forms create GUIs. A form is a graphical element that appears on the desktop. A form can be a dialog or a window.
- A component is a class that implements the **`IComponent`** interface.
- A control is a graphical component, such as a button. Components that are not visible usually are referred to simply as components.
- The active window has the focus. It is the frontmost window and has a highlighted title bar.
- A form acts as a container for components.
- When the user interacts with a control, an event is generated. This event can trigger methods that respond to the user’s actions.
- All forms, components and controls are classes.
- The general design process for creating Windows applications involves creating a Windows Form, setting its properties, adding controls, setting their properties and configuring event handlers.
- GUIs are event driven. When a user interaction occurs, an event is generated. The event information then is passed to event handlers.
- Events are based on the notion of delegates. Delegates act as an intermediate step between the object creating (raising) the event and the method handling it.
- In many cases, the programmer will handle events generated by prepackaged controls. In this case, all the programmer needs to do is create and register the event handler.
- Use the **Events** window to create and register event handlers.
- The information we need to register an event is the **`EventArgs`** class (to define the event handler) and the **`EventHandler`** delegate (to register the event handler). Visual Studio .NET can usually register the event for us.
- Labels (class **`Label`**) display read-only text instructions or information on a GUI.
- A **`TextBox`** is a single-line area in which text can be entered. A password text box displays only a certain character (such as **`*`**) when text is input.
- A **`Button`** is a control that the user clicks to trigger a specific action. Buttons typically respond to the **`Click`** event.
- **`GroupBox`**es and **`Panel`**s help arrange components on a GUI. The main difference between the classes is that **`GroupBox`**es can display text, and **`Panel`**s can have scrollbars.
- Visual C# has two types of state buttons—**`CheckBox`**es and **`RadioButton`**s—that have on/off or true/false values.
- A checkbox is a small white square that can be blank or contain a checkmark.
- Use the bitwise XOR operator (**`^`**) to combine or negate a font style.
- Radio buttons (class **`RadioButton`**) have two states—selected and not selected. Radio buttons appear as a group in which only one radio button can be selected at a time. To create new groups, radio buttons must be added to **`GroupBox`**es or **`Panel`**s. Each **`GroupBox`** or **`Panel`** is a group.
- Radio buttons and checkboxes use the **`CheckChanged`** event.

- Scrollbars are controls that allow the user to access a range of integer values. There are horizontal **`HScrollBar`**s and vertical **`VScrollBar`**s. Whenever a scrollbar is changed, it raises a **`Scroll`** event.
- A picture box (class **`PictureBox`**) displays an image (set by an object of class **`Image`**).
- Mouse events (clicks, presses and moves) can be handled for any GUI control that derives from **`System.Windows.Forms.Control`**. Mouse events use class **`MouseEventArgs`** (**`MouseEventHandler`** delegate) and **`EventArgs`** (**`EventHandler`** delegate).
- Class **`MouseEventArgs`** contains information about the *x*- and *y*-coordinates, the button used, the number of clicks and the number of notches through which the mouse wheel turned.
- Key events are generated when keyboard's keys are pressed and released. These events can be handled by any control that inherits from **`System.Windows.Forms.Control`**.
- Event **`KeyPress`** can return a **`char`** for any ASCII character pressed. One cannot determine if special modifier keys (such as *Shift*, *Alt* and *Control*) were pressed.
- Events **`KeyUp`** and **`KeyDown`** test for special modifier keys (using **`KeyEventArgs`**). The delegates are **`KeyPressEventHandler`** (**`KeyPressEventArgs`**) and **`KeyEventHandler`** (**`KeyEventArgs`**).
- Class **`KeyEventArgs`** has properties **`KeyCode`**, **`KeyData`** and **`KeyValue`**.
- Property **`KeyCode`** returns the key pressed, but does not give any information about modifier keys.
- The **`KeyData`** property includes data about modifier keys.
- The **`KeyValue`** property returns the key code for the key pressed as an integer.

TERMINOLOGY

active window
`Alt` property
ASCII character
background color
button
`Button` class
button label
checkbox
`CheckBox` class
checkbox label
`CheckedChanged` event
click a button
`Click` event
click a mouse button
component
container
control
`Control` property
delegate
deselected
drag and drop
`#endregion`
Enter key
`Enter` mouse event
event
event argument
event delegate
event driven
event handler
event-handling model
`event` keyword
`EventArgs` class
Events window in Visual Studio .NET
focus
`Font` property
font style
form
`Form` class
`GetCurrentDirectory` method
graphical user interface (GUI)
`GroupBox`
GUI component
handle event
`HScrollBar` class
`Image` property
`InitializeComponent` method
input data from the keyboard
key code
key data
key event

key value
keyboard
`KeyDown` event
`KeyEventArgs` class
`KeyPress` event
`KeyPressEventArgs` class
`KeyUp` event
label
`Label` class
list
menu
menu bar
mouse
mouse click
mouse event
mouse move
mouse press
`MouseDown` event
`MouseEventArgs` class
`MouseEventHandler` delegate
`MouseHover` event
`MouseLeave` event
`MouseMove` event
`MouseUp` event
`MouseWheel` event
moving the mouse
multicast
multicast event
`MulticastDelegate` class
mutual exclusion
`Name` property
`NewValue` property
panel
`Panel` class
password box
`PasswordChar` property
picture box
`PictureBox` class
radio button
radio-button group
`RadioButton` class
raise an event
read-only text
`#region` tag
register an event handler
`Scroll` event
scrollbar
scrollbar in panel
`ScrollEventArgs` class
selecting an item from a menu
`Shift` property
`SizeMode` property
`System.Windows.Forms` namespace
`Text` property
text box
`TextBox` class
`TextChanged` event
trigger an event
type in a textbox
uneditable text or icon
virtual key code
visual programming
`VScrollBar` class
widget
window gadget
Windows Form
XOR

SELF-REVIEW EXERCISES

12.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) A GUI is a pictorial interface to a program.
b) Windows Forms commonly are used to create GUIs.
c) A control is a nonvisible component.
d) All forms, components and controls are classes.
e) Events are based on properties.
f) Class **`Label`** is used to provide pictorial instructions or information.
g) Button presses raise events.
h) Checkboxes in the same group are mutually exclusive.
i) Scrollbars allow the user to maximize or minimize a set of data.
j) All mouse events use the same event arguments class.
k) Visual Studio .NET can register an event and create an empty event handler.

12.2 Fill in the blanks in each of the following statements:

a) The active window is said to have the __________.

b) The form acts as a __________ for the components that are added.

c) GUIs are __________ driven.

d) Every method that handles the same event must have the same __________.

e) The information required when registering an event handler is the__________ class and the __________.

f) A __________ textbox displays only a single character (such as an asterisk) as the user types.

g) Class __________ and class __________ help arrange components on a GUI and provide logical group for radio buttons.

h) Typical mouse events include __________, __________ and __________.

i) __________ events are generated when a key on the keyboard is pressed or released.

j) The modifier keys are __________, __________ and __________.

k) A __________ event or delegate can call multiple methods.

ANSWERS TO SELF-REVIEW EXERCISES

12.1 a) True. b) True. c) False. A control is a visible component. d) True. e) False. Events are based on delegates. f) False. Class `Label` is used to provide text instructions or information. g) True. h) False. Radio buttons in the same group are mutually exclusive. i) False. Scrollbars allow the user to view data that normally cannot fit in its container. j) False. Some mouse events use `EventArgs`, while others use `MouseEventArgs`. k) True.

12.2 a) focus. b) container. c) event. d) signature. e) event arguments, delegate. f) password. g) `GroupBox`, `Panel`. h) mouse clicks, mouse presses, mouse moves. i) Key. j) *Shift*, *Control*, *Alt*. k) multicast.

EXERCISES

12.3 Extend the program in Fig. 12.24 to include a `CheckBox` for every font style option. [*Hint*: Use XOR rather than testing for every bit explicitly.]

12.4 Create the GUI in Fig. 12.33. You do not have to provide any functionality.

12.5 Create the GUI in Fig. 12.34. You do not have to provide any functionality.

12.6 Extend the program of Fig. 12.30 to include options for changing the size and color of the lines drawn. Create a GUI similar to the one in Fig. 12.35. [*Hint*: Have variables to keep track of the currently selected size (`int`) and color (`Color` object). Set them using the event handlers for the radio buttons. For the color, use the various `Color` constants (such as `Color.Blue`). When responding to the mouse moves, simply use the size and color variables to determine the proper size and color.]

12.7 Write a program that plays "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1–1000. The program then displays the following text in a label:

```
I have a number between 1 and 1000—can you guess my number?
Please enter your first guess.
```

A `TextBox` should be used to input the guess. As each guess is input, the background color should change to either red or blue. Red indicates that the user is getting "warmer," and blue indicates that the user is getting "colder." A `Label` should display either "`Too High`" or "`Too Low`" to help the user choose a number closer toward the correct answer. When the user obtains the correct answer, "**`Correct!`**" should be displayed. The background should become green and the **`TextBox`** used for input should become uneditable. Provide a **`Button`** that allows the user to play the game again. When the **`Button`** is clicked, generate a new random number, change the background to the default color and generate the input **`TextBox`** to editable.

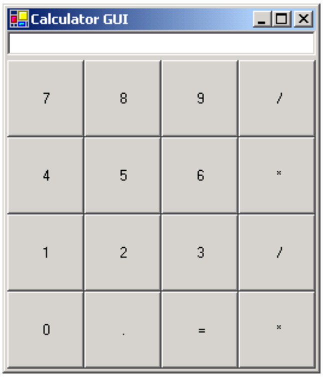
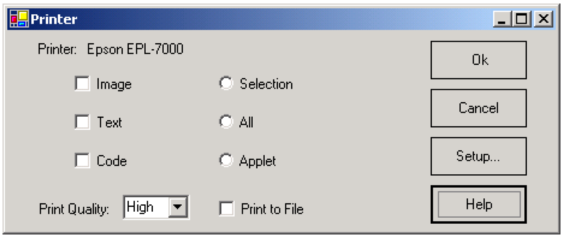

Calculator GUI
7
8
9
/
4
5
6
*
1
2
3
/
0
.
=
*

Fig. 12.33 GUI for Exercise 12.4.


Printer
Image
Selection
Text
All
Code
Applet
Print to File
Ok
Cancel
Setup...
Help

Printer: Epson EPL-7000

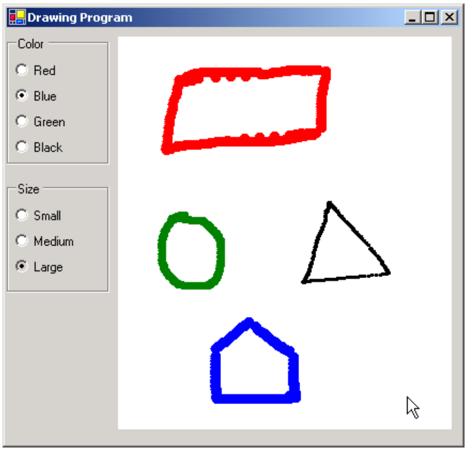
☐ Image ○ Selection

☐ Text ○ All

☐ Code ○ Applet

Print Quality: High ☐ Print to File

Fig. 12.34 GUI for Exercise 12.5.


Drawing Program
Color
Size

○ Red
● Blue
○ Green
○ Black

○ Small
○ Medium
● Large

Fig. 12.35 GUI for Exercise 12.6.

13

Graphical User Interfaces Concepts: Part 2

Objectives

- To be able to create menus, window tabs and multiple-document-interface (MDI) programs.
- To understand the use of the **`ListView`** and **`TreeView`** controls for displaying information.
- To be able to use hyperlinks with the **`LinkLabel`** control.
- To be able to display lists using **`ListBox`**es and **`ComboBox`**es.
- To create custom controls.

I claim not to have controlled events, but confess plainly that events have controlled me.
Abraham Lincoln

A good symbol is the best argument, and is a missionary to persuade thousands.
Ralph Waldo Emerson

Capture its reality in paint!
Paul Cézanne

But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun!
William Shakespeare

An actor entering through the door, you've got nothing. But if he enters through the window, you've got a situation.
Billy Wilder

Outline

13.1 Introduction
13.2 Menus
13.3 `LinkLabels`
13.4 `ListBoxes` and `CheckedListBoxes`
  13.4.1 `ListBoxes`
  13.4.2 `CheckedListBoxes`
13.5 `ComboBoxes`
13.6 `TreeViews`
13.7 `ListViews`
13.8 Tab Control
13.9 Multiple-Document-Interface (MDI) Windows
13.10 Visual Inheritance
13.11 User-Defined Controls

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

13.1 Introduction

This chapter continues our study of GUIs. We begin our discussion of more advanced topics with a commonly used GUI component, the *menu*, which presents a user with several logically organized options. The reader will learn how to develop menus with the tools provided by Visual Studio .NET. We introduce **`LinkLabel`**s, powerful GUI components that enable the user to click the mouse to be taken to one of several destinations.

We consider GUI components that encapsulate smaller GUI components. We demonstrate how to manipulate a list of values via a **`ListBox`** and how to combine several checkboxes in a **`CheckedListBox`**. We also create drop-down lists using **`ComboBox`**es and display data hierarchically with a **`TreeView`** control. We present two important GUI components—tab controls and multiple-document-interface windows. These components enable developers to create real-world programs with sophisticated graphical user interfaces.

Most of the GUI components used in this book are included with Visual Studio .NET. We show how to design custom controls and add those controls to the **ToolBox**. The techniques in this chapter form the groundwork for the creation of complex GUIs and custom controls.

13.2 Menus

Menus are used to provide groups of related commands for Windows applications. Although these commands depend on the program, some—such as **Open** and **Save**—are common to many applications. Menus are an integral part of GUIs, because they make user actions possible without unnecessary “cluttering” of GUIs.

In Fig. 13.1, an expanded menu lists various commands (called *menu items*), plus *submenus* (menus within a menu). Notice that the top-level menus appear in the left portion of the figure, whereas any submenus or menu items are displayed to the right. The menu that contains a menu item is called that menu item's *parent menu*. A menu item that contains a submenu is considered to be the parent of that submenu.

All menu items can have *Alt* key shortcuts (also called *access shortcuts* or *hot keys*), which are accessed by pressing *Alt* and the underlined letter (for example, *Alt* + *F* retrieves the **File** menu). Menus that are not top-level menus can have shortcut keys as well (combinations of *Ctrl*, *Shift*, *Alt*, *F1*, *F2*, letter keys etc.). Some menu items display checkmarks, usually indicating that multiple options on the menu can be selected at once.

To create a menu, open the **Toolbox**, and drag a **`MainMenu`** control onto the form. This creates a menu bar on the top of the form and places a **`MainMenu`** icon underneath it. To select the **`MainMenu`**, click the icon. This setup is known as the Visual Studio .NET **Menu Designer**, which allows the user to create and edit menus. Menus are like other controls; they have properties, which can be accessed through the **Properties** window or the **Menu Designer** (Fig. 13.2), and events, which can be accessed through the **Class Name** and **Method Name** drop-down menus.

Look-and-Feel Observation 13.1

***`Button`s** also can have access shortcuts. Place the **`&`** symbol just before the character via which we wish to create a shortcut. To click the button, the user then presses* Alt *and the underlined character.*

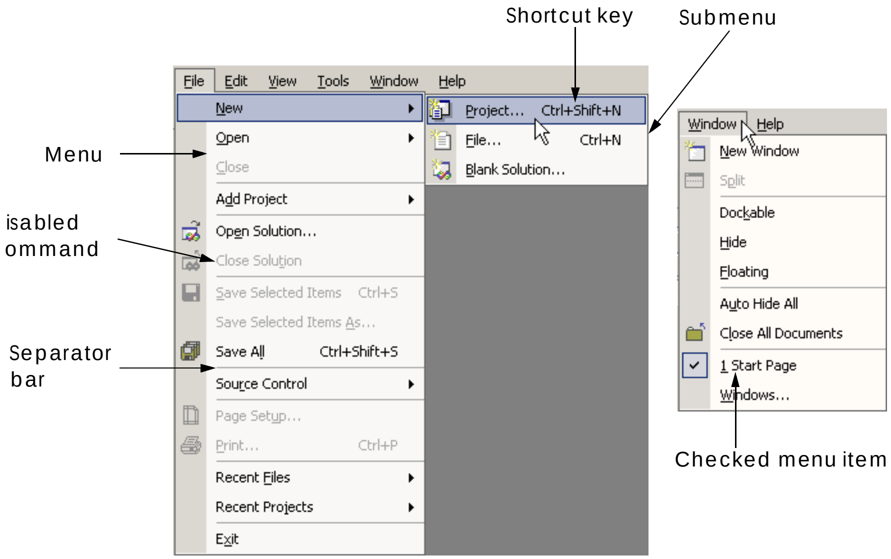

Shortcut key
Submenu
Menu
isabled
ommand
Separator
bar
Checked menu item
File
Edit
View
Tools
Window
Help
New
Open
Close
Add Project
Open Solution...
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Source Control
Page Setup...
Print... Ctrl+P
Recent Files
Recent Projects
Exit
Project... Ctrl+Shift+N
File... Ctrl+N
Blank Solution...
Window
Help
New Window
Split
Dockable
Hide
Floating
Auto Hide All
Close All Documents
1 Start Page
Windows...

Fig. 13.1 Expanded and checked menus.

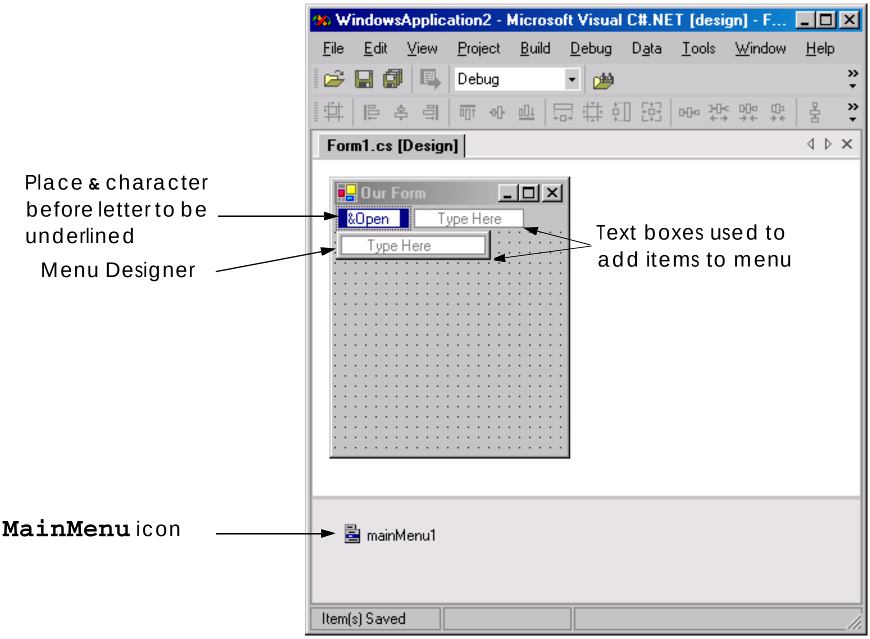

WindowsApplication2 - Microsoft Visual C#.NET [design] - F...
File Edit View Project Build Debug Data Tools Window Help
Debug
Form1.cs [Design]
Our Form
&Open
Type Here
Type Here
mainMenu1
Item(s) Saved
Place & character before letter to be underlined
Menu Designer
Text boxes used to add items to menu
`MainMenu` icon

Fig. 13.2 Visual Studio .NET **Menu Designer**.

To add entries to the menu, click the **Type Here** textbox and type the text that should appear in the menu. Each entry in the menu is of type ***`MenuItem`*** from the **`System.Windows.Forms`** namespace. The menu itself is of type ***`MainMenu`***. After the programmer presses the *Enter* key, the menu item is added. Then, more **Type Here** textboxes appear, allowing us to add items underneath or to the side of the original menu item (Fig. 13.2). To create an access shortcut, type an ampersand (**`&`**) in front of the character to be underlined. For example, to create the **File** menu item, type **`&File`**. The actual ampersand character is displayed by typing **`&&`**. To add other shortcut keys (such as *Ctrl + F9*), set the **`Shortcut`** property of the **`MenuItem`**.

Programmers can remove a menu item by selecting it with the mouse and pressing the *Delete* key. Separator bars are inserted by right-clicking the menu and selecting **Insert Separator** or by typing "**`-`**" as the menu text.

Menu items generate a **`Click`** event when selected. To create an empty event handler, enter code-view mode, double click on the **`MenuItem`** in design view. Menus can also display the names of open windows in multiple-document-interface (MDI) forms (see Section 13.9). Menu properties and events are summarized in Fig. 13.3.

Look-and-Feel Observation 13.2

It is conventional to place an ellipsis (...) after a menu item that brings up a dialog (such as ***Save As...****). Menu items that produce an immediate action without prompting the user (such as* ***Save****) should not have an ellipsis following their name.*

Class **`MenuTest`** (Fig. 13.4) creates a simple menu on a form. The form has a top-level **File** menu with menu items **About** (displays a message box) and **Exit** (terminates the program).The menu also includes a **Format** menu, which changes the text on a label. The **Format** menu has submenus **Color** and **Font**, which change the color and font of the text on a label.

Look-and-Feel Observation 13.3

Using common Windows shortcuts (such as Ctrl+F for Find operations and Ctrl+S for Save operations) decreases an application's learning curve.

We begin by dragging the **`MainMenu`** from the **ToolBox** onto the form. We then create our entire menu structure, using the **Menu Designer**. The **File** menu has items **About** (**`aboutMenuItem`**, line 21) and **Exit** (**`exitMenuItem`**, line 22); the **Format** menu (**`formatMenu`**, line 25) has two submenus. The first submenu, **Color** (**`colorMenuItem`**, line 28), contains menu items **Black** (**`blackMenuItem`**, line 29), **Blue** (**`blueMenuItem`**, line 30), **Red** (**`redMenuItem`**, line 31) and **Green** (**`greenMenuItem`**, line 32). The second submenu, **Font** (**`fontMenuItem`**, line 40), contains menu items **Times New Roman** (**`timesMenuItem`**, line 35), **Courier** (**`courierMenuItem`**, line 36), **Comic Sans** (**`comicMenuItem`**, line 37), a separator bar (**`separatorMenuItem`**, line 42), **Bold** (**`boldMenuItem`**, line 38) and **Italic** (**`italicMenuItem`**, line 39).

`MainMenu` and `MenuItem` events and properties	**Description / Delegate and Event Arguments**
***`MainMenu`* Properties**	
`MenuItems`	Lists the **`MenuItem`**s that are contained in the **`MainMenu`**.
`RightToLeft`	Causes text to display from right to left. Useful for languages that are read from right to left.
***`MenuItem`* Properties**	
`Checked`	Indicates whether a menu item is checked (according to property **`RadioCheck`**). Default **`False`**, meaning that the menu item is not checked.
`Index`	Specifies an item's position in its parent menu.
`MenuItems`	Lists the submenu items for a particular menu item.
`MergeOrder`	Sets the position of a menu item when its parent menu is merged with another menu.
`MergeType`	Takes a value of the **`MenuMerge`** enumeration. Specifies how a parent menu merges with another menu. Possible values are **`Add`**, **`MergeItems`**, **`Remove`** and **`Replace`**.
`RadioCheck`	Indicates whether a selected menu item appears as a radio button (black circle) or displays a checkmark. **`True`** creates radio button, **`False`** displays checkmark; default **`False`**.
`Shortcut`	Specifies the shortcut key for the menu item (e.g., *Ctrl* + *F9* can be equivalent to clicking a specific item).

Fig. 13.3 **`MainMenu`** and **`MenuItem`** properties and events. (Part 1 of 2.)

`MainMenu` and `MenuItem` events and properties	**Description / Delegate and Event Arguments**
`ShowShortcut`	Indicates whether a shortcut key is shown beside menu item text. Default is **`True`**, which displays the shortcut key.
`Text`	Specifies the text to appear in the menu item. To create an *Alt* access shortcut, precede a character with **`&`** (e.g., **`&File`** for **File**).
Common Event	*(Delegate **`EventHandler`**, event arguments **`EventArgs`**)*
`Click`	Generated when item is clicked or shortcut key is used. Default when double-clicked in designer.

Fig. 13.3 **`MainMenu`** and **`MenuItem`** properties and events. (Part 2 of 2.)

```
// Fig 13.4: MenuTest.cs
// Using menus to change font colors and styles.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class MenuTest : System.Windows.Forms.Form
{
   // display label
   private System.Windows.Forms.Label displayLabel;

   // main menu (contains file and format menu)
   private System.Windows.Forms.MainMenu mainMenu;

   // file menu
   private System.Windows.Forms.MenuItem fileMenuItem;
   private System.Windows.Forms.MenuItem aboutMenuItem;
   private System.Windows.Forms.MenuItem exitMenuItem;

   // format menu
   private System.Windows.Forms.MenuItem formatMenuItem;

   // color submenu
   private System.Windows.Forms.MenuItem colorMenuItem;
   private System.Windows.Forms.MenuItem blackMenuItem;
   private System.Windows.Forms.MenuItem blueMenuItem;
   private System.Windows.Forms.MenuItem redMenuItem;
   private System.Windows.Forms.MenuItem greenMenuItem;

   // font submenu
   private System.Windows.Forms.MenuItem timesMenuItem;
```

Fig. 13.4 Menus for changing text font and color. (Part 1 of 5.)

```
      private System.Windows.Forms.MenuItem courierMenuItem;
      private System.Windows.Forms.MenuItem comicMenuItem;
      private System.Windows.Forms.MenuItem boldMenuItem;
      private System.Windows.Forms.MenuItem italicMenuItem;
      private System.Windows.Forms.MenuItem fontMenuItem;

      private System.Windows.Forms.MenuItem separatorMenuItem;

      [STAThread]
      static void Main()
      {
         Application.Run( new MenuTest() );
      }

      // display MessageBox
      private void aboutMenuItem_Click(
         object sender, System.EventArgs e )
      {
         MessageBox.Show(
            "This is an example\nof using menus.",
            "About", MessageBoxButtons.OK,
            MessageBoxIcon.Information );
      }

      // exit program
      private void exitMenuItem_Click(
         object sender, System.EventArgs e )
      {
         Application.Exit();
      }

      // reset color
      private void ClearColor()
      {
         // clear all checkmarks
         blackMenuItem.Checked = false;
         blueMenuItem.Checked = false;
         redMenuItem.Checked = false;
         greenMenuItem.Checked = false;
      }

      // update menu state and color display black
      private void blackMenuItem_Click(
         object sender, System.EventArgs e )
      {
         // reset checkmarks for color menu items
         ClearColor();

         // set color to black
         displayLabel.ForeColor = Color.Black;
         blackMenuItem.Checked = true;
      }

```

Fig. 13.4 Menus for changing text font and color. (Part 2 of 5.)

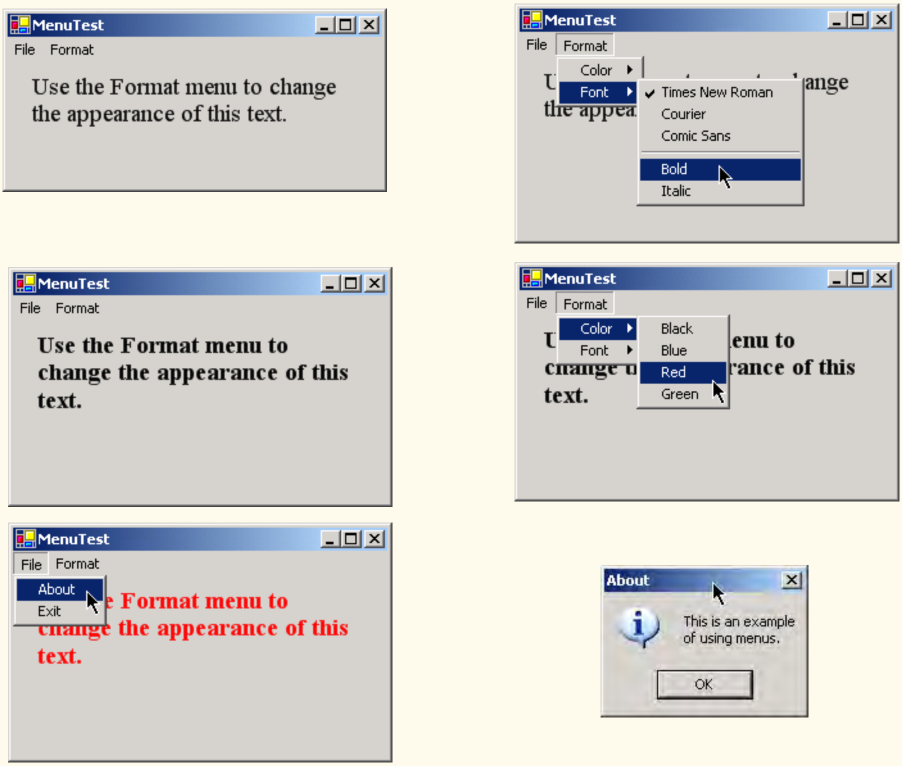
```
      // update menu state and color display blue
      private void blueMenuItem_Click(
         object sender, System.EventArgs e )
      {
         // reset checkmarks for color menu items
         ClearColor();

         // set color to blue
         displayLabel.ForeColor = Color.Blue;
         blueMenuItem.Checked = true;
      }

      // update menu state and color display red
      private void redMenuItem_Click(
         object sender, System.EventArgs e )
      {
         // reset checkmarks for color menu items
         ClearColor();

         // set color to red
         displayLabel.ForeColor = Color.Red;
         redMenuItem.Checked = true;
      }

      // update menu state and color display green
      private void greenMenuItem_Click(
         object sender, System.EventArgs e )
      {
         // reset checkmarks for color menu items
         ClearColor();

         // set color to green
         displayLabel.ForeColor = Color.Green;
         greenMenuItem.Checked = true;
      }

      // reset font types
      private void ClearFont()
      {
         // clear all checkmarks
         timesMenuItem.Checked = false;
         courierMenuItem.Checked = false;
         comicMenuItem.Checked = false;
      }

      // update menu state and set font to Times
      private void timesMenuItem_Click(
         object sender, System.EventArgs e )
      {
         // reset checkmarks for font menu items
         ClearFont();

```

Fig. 13.4 Menus for changing text font and color. (Part 3 of 5.)

```
      // set Times New Roman font
      timesMenuItem.Checked = true;
      displayLabel.Font = new Font(
         "Times New Roman", 14, displayLabel.Font.Style );
   }

   // update menu state and set font to Courier
   private void courierMenuItem_Click(
      object sender, System.EventArgs e )
   {
      // reset checkmarks for font menu items
      ClearFont();

      // set Courier font
      courierMenuItem.Checked = true;
      displayLabel.Font = new Font(
         "Courier New", 14, displayLabel.Font.Style );
   }

   // update menu state and set font to Comic Sans MS
   private void comicMenuItem_Click(
      object sender, System.EventArgs e )
   {
      // reset checkmarks for font menu items
      ClearFont();

      // set Comic Sans font
      comicMenuItem.Checked = true;
      displayLabel.Font = new Font(
         "Comic Sans MS", 14, displayLabel.Font.Style );
   }

   // toggle checkmark and toggle bold style
   private void boldMenuItem_Click(
      object sender, System.EventArgs e )
   {
      // toggle checkmark
      boldMenuItem.Checked = !boldMenuItem.Checked;

      // use Xor to toggle bold, keep all other styles
      displayLabel.Font = new Font(
         displayLabel.Font.FontFamily, 14,
         displayLabel.Font.Style ^ FontStyle.Bold );
   }

   // toggle checkmark and toggle italic style
   private void italicMenuItem_Click(
      object sender, System.EventArgs e)
   {
      // toggle checkmark
      italicMenuItem.Checked = !italicMenuItem.Checked;

```

Fig. 13.4 Menus for changing text font and color. (Part 4 of 5.)

```
193       // use Xor to toggle bold, keep all other styles
194       displayLabel.Font = new Font(
195          displayLabel.Font.FontFamily, 14,
196          displayLabel.Font.Style ^ FontStyle.Italic );
197    }
198
199 } // end class MenuTest
```



Use the Format menu to change the appearance of this text.

About

This is an example of using menus.

Fig. 13.4 Menus for changing text font and color. (Part 5 of 5.)

The **About** menu item in the **File** menu displays a **`MessageBox`** when clicked (lines 54–57). The **Exit** menu item closes the application through **`static`** method ***`Exit`*** of class ***`Application`*** (line 64). Class **`Application`** contains **`static`** methods used to control program execution. Method **`Exit`** causes our application to quit.

We made the items in the **Color** submenu (**Black**, **Blue**, **Red** and **Green**) mutually exclusive—the user can select only one at a time (we explain how we did this shortly). To indicate this behavior to the user, we set the menu item's ***`RadioCheck`*** properties to **`True`**. This causes a radio button to appear (instead of a checkmark) when a user selects a color-menu item.

Each **Color** menu item has its own event handler. The event handler for color **Black** is **`blackMenuItem_Click`** (lines 78–87). The event handlers for colors **Blue**, **Red** and **Green** are **`blueMenuItem_Click`** (lines 90–99), **`redMenuItem_Click`** (lines 102–111) and **`greenMenuItem_Click`** (lines 114–123), respectively. Each **Color** menu item must be mutually exclusive, so each event handler calls method `ClearColor` (lines 68–75) before setting its corresponding `Checked` property to `True`. Method `ClearColor` sets the `Checked` property of each color `MenuItem` to `False`, effectively preventing more than one menu item from being checked at a time.

Software Engineering Observation 13.1

The mutual exclusion of menu items is not enforced by the `MainMenu`*, even when the* `RadioCheck` *property is* `True`*. We must program this behavior.*

Look-and-Feel Observation 13.4

Set the *`RadioCheck`* *property to reflect the desired behavior of menu items. Use radio buttons (*`RadioCheck` *property set to* `True`*) to indicate mutually exclusive menu items. Use check marks (*`RadioCheck` *property set to* `False`*) for menu items that have no logical restriction.*

The **Font** menu contains three menu items for font types (**Courier**, **Times New Roman** and **Comic Sans**) and two menu items for font styles (**Bold** and **Italic**). We add a separator bar between the font-type and font-style menu items to indicate the distinction: Font types are mutually exclusive, but styles are not. This means that a `Font` object can specify only one font face at a time, but can set multiple styles at once (e.g., a font can be both bold and italic). We set the font-type menu items to display checks. As with the **Color** menu, we also must enforce mutual exclusion in our event handlers.

Event handlers for font-type menu items **TimesRoman**, **Courier** and **ComicSans** are `timesMenuItem_Click` (lines 135–145), `courierMenuItem_Click` (lines 148–158) and `comicMenuItem_Click` (lines 161–171), respectively. These event handlers behave in a manner similar to that of the event handlers for the **Color** menu items. Each event handler clears the `Checked` properties for all font-type menu items by calling method `ClearFont` (lines 126–132), then sets the `Checked` property of the menu item that generated the event to `True`. This enforces the mutual exclusion of the font-type menu items.

The event handlers for the **Bold** and **Italic** menu items (lines 174–197) use the bitwise `Xor` operator. For each font style, the exclusive or operator (^) changes the text to include the style or, if that style is already applied, to remove it. The toggling behavior provided by the `Xor` operator is explained in Chapter 12, Graphical User Interfaces: Part 1. As explained in Chapter 12, this program's event-handling structure allows us to add and remove menu entries while making minimal structural changes to the code.

13.3 LinkLabels

The *`LinkLabel`* control displays links to other objects, such as files or Web pages (Fig. 13.5). A `LinkLabel` appears as underlined text (colored blue by default). When the mouse moves over the link, the pointer changes to a hand; this is similar to the behavior of a hyperlink in a Web page. The link can change color to indicate whether the link is new, visited or active. When clicked, the `LinkLabel` generates a `LinkClicked` event (see Fig. 13.6). Class `LinkLabel` is derived from class `Label` and therefore inherits all of class `Label`'s functionality.

Class `LinkLabelTest` (Fig. 13.7) uses three `LinkLabel`s, to link to the `C:\` drive, the Deitel Web page (`www.deitel.com`) and the Notepad application, respectively. The `Text` properties of the `LinkLabel`s `driveLinkLabel` (line 14), `deitelLinkLabel` (line 15) and `notepadLinkLabel` (line 16) are set to describe each link's purpose.

Look-and-Feel Observation 13.5

*Although other controls can perform actions similar to those of a **`LinkLabel`** (such as the opening of a Web page), **`LinkLabel`**s indicate that a link can be followed—a regular label or button does not necessarily convey that idea.*

The event handlers for the **`LinkLabel`** instances call static method ***`Start`*** of class ***`Process`*** (namespace **`System.Diagnostics`**). This method allows us to execute other programs from our application. Method **`Start`** can take as arguments either the file to open (a **`String`**) or the name of the application to run and its command-line arguments (two **`String`**s). Method **`Start`**'s arguments can be in the same form as if they were provided for input to the **Run** command in Windows. To open a file that has a file type that Windows recognizes, simply insert the file's full path name. The Windows operating system should be able to use the application associated with the given file's extension to open the file.

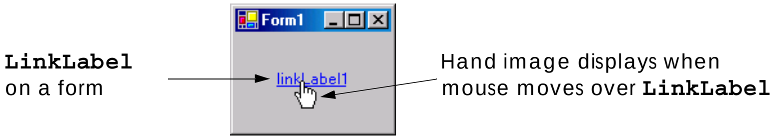


Fig. 13.5 **`LinkLabel`** control in the design phase and in running program.

`LinkLabel` **properties and events**	**Description / Delegate and Event Arguments**
Common Properties	
`ActiveLinkColor`	Specifies the color of the active link when clicked. Default is red.
`LinkArea`	Specifies which portion of text in the **`LinkLabel`** is treated as part of the link.
`LinkBehavior`	Specifies the link's behavior, such as how the link appears when the mouse is placed over it.
`LinkColor`	Specifies the original color of all links before they have been visited. Default is blue.
`Links`	Lists the **`LinkLabel.Link`** objects, which are the links contained in the **`LinkLabel`**.
`LinkVisited`	If **`True`**, link appears as if it were visited (its color is changed to that specified by property **`VisitedLinkColor`**). Default **`False`**.
`Text`	Specifies the text to appear on the control.
`UseMnemonic`	If **`True`**, **`&`** character in **`Text`** property acts as a shortcut (similar to the *Alt* shortcut in menus).
`VisitedLinkColor`	Specifies the color of visited links. Default is **`Color.Purple`**.

Fig. 13.6 **`LinkLabel`** properties and events. (Part 1 of 2.)

`LinkLabel` **properties and events**	**Description / Delegate and Event Arguments**
Common Event	*(Delegate* ***`LinkLabelLinkClickedEventHandler`****, event arguments* ***`LinkLabelLinkClickedEventArgs`****)*
`LinkClicked`	Generated when link is clicked. Default when control is double-clicked in designer.

Fig. 13.6 **`LinkLabel`** properties and events. (Part 2 of 2.)

The event handler for **`driveLinkLabel`**’s **`LinkClicked`** events browses the **`C:\`** drive (lines 25–30). Line 28 sets the **`LinkVisited`** property to **`True`**, which changes the link’s color from blue to purple (we can configure the **`LinkVisited`** colors through the **Properties** window in the Visual Studio .NET IDE). The event handler then passes **`"C:\"`** to method **`Start`** (line 29), which opens a **Windows Explorer** window.

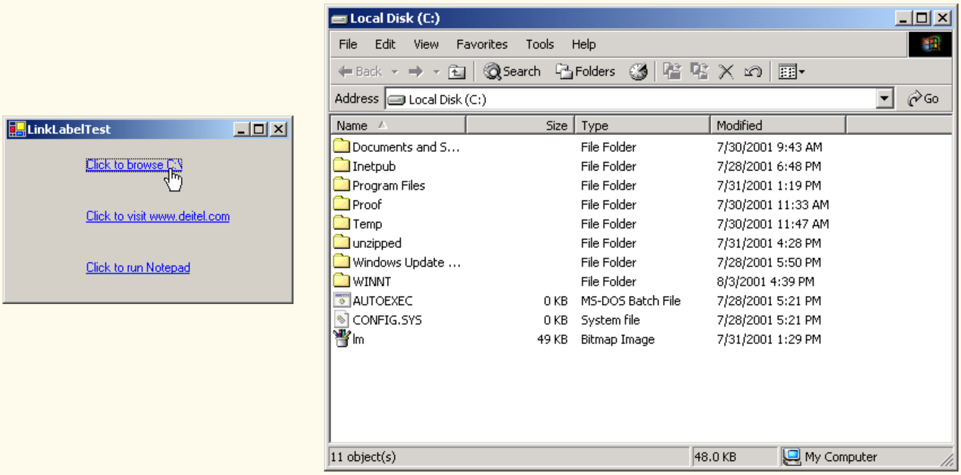
The event handler for **`deitelLinkLabel`**’s **`LinkClicked`** events (lines 33–39) opens the Web page **`www.deitel.com`** in Internet Explorer. We achieve this by passing the string **`"IExplore"`** and the Web-page address (lines 37–38), which opens Internet Explorer. Line 36 sets the **`LinkVisited`** property to **`True`**.

The event handler for **`notepadLinkLabel`**’s **`LinkClicked`** events opens the specified Notepad application (lines 42–51). Line 46 sets the link to appear as a visited link. Line 50 passes the argument **`"notepad"`** to method **`Start`**, which calls **`notepad.exe`**. Note that, in line 50, the **`.exe`** extension is not required—Windows can determine whether the argument given to method **`Start`** is an executable file.

```
1  // Fig. 13.7: LinkLabelTest.cs
2  // Using LinkLabels to create hyperlinks.
3  
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Data;
10 
11 public class LinkLabelTest : System.Windows.Forms.Form
12 {
13    // linklabels to C: drive, www.deitel.com and Notepad
14    private System.Windows.Forms.LinkLabel driveLinkLabel;
15    private System.Windows.Forms.LinkLabel deitelLinkLabel;
16    private System.Windows.Forms.LinkLabel notepadLinkLabel;
17 
18    [STAThread]
19    static void Main()
20    {
21       Application.Run( new LinkLabelTest() );
22    }
```

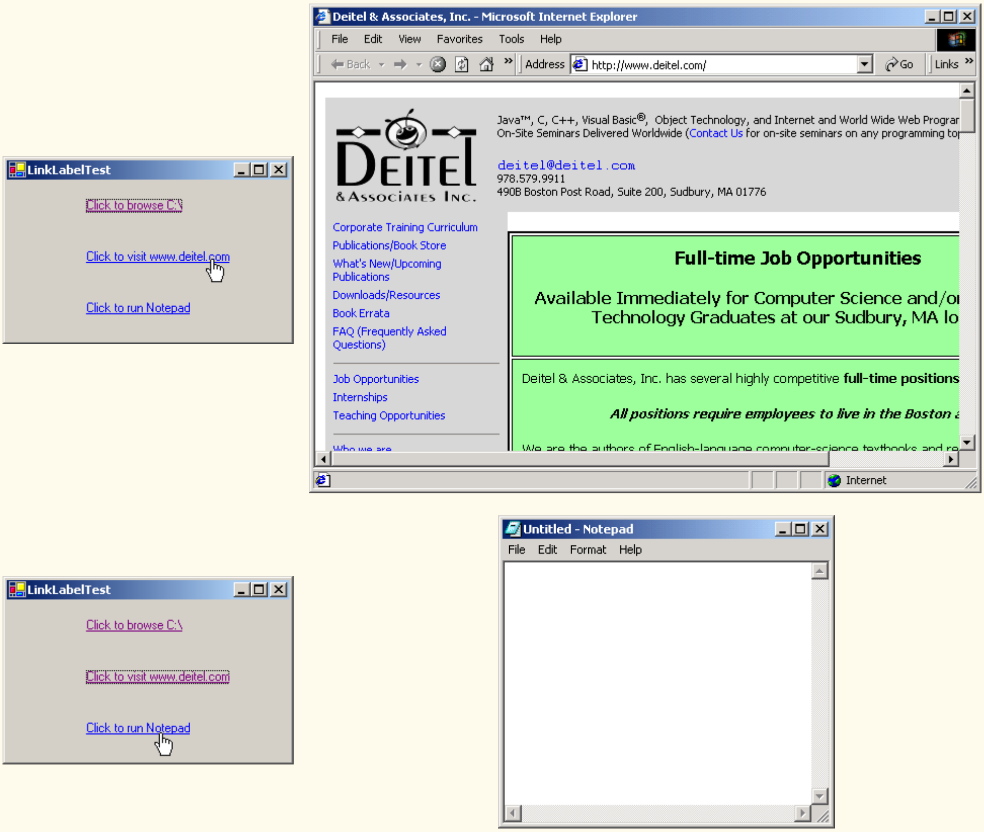
Fig. 13.7 **`LinkLabel`**s used to link to a folder, a Web page and an application. (Part 1 of 3.)

```

   // browse C:\ drive
   private void driveLinkLabel_LinkClicked( object sender,
      System.Windows.Forms.LinkLabelLinkClickedEventArgs e )
   {
      driveLinkLabel.LinkVisited = true;
      System.Diagnostics.Process.Start( "C:\\" );
   }

   // load www.deitel.com in Web broswer
   private void deitelLinkLabel_LinkClicked( object sender,
      System.Windows.Forms.LinkLabelLinkClickedEventArgs e )
   {
      deitelLinkLabel.LinkVisited = true;
      System.Diagnostics.Process.Start(
         "IExplore", "http://www.deitel.com" );
   }

   // run application Notepad
   private void notepadLinkLabel_LinkClicked(
      object sender,
      System.Windows.Forms.LinkLabelLinkClickedEventArgs e )
   {
      notepadLinkLabel.LinkVisited = true;

      // program called as if in run
      // menu and full path not needed
      System.Diagnostics.Process.Start( "notepad" );
   }

} // end class LinkLabelTest
```


LinkLabelTest
Click to browse C:\
Click to visit www.deitel.com
Click to run Notepad
Local Disk (C:)
File Edit View Favorites Tools Help
Back Search Folders
Address Local Disk (C:)
Go
11 object(s)
48.0 KB
My Computer

Name	Size	Type	Modified
Documents and S...		File Folder	7/30/2001 9:43 AM
Inetpub		File Folder	7/28/2001 6:48 PM
Program Files		File Folder	7/31/2001 1:19 PM
Proof		File Folder	7/30/2001 11:33 AM
Temp		File Folder	7/30/2001 11:47 AM
unzipped		File Folder	7/31/2001 4:28 PM
Windows Update ...		File Folder	7/28/2001 5:50 PM
WINNT		File Folder	8/3/2001 4:39 PM
AUTOEXEC	0 KB	MS-DOS Batch File	7/28/2001 5:21 PM
CONFIG.SYS	0 KB	System file	7/28/2001 5:21 PM
lm	49 KB	Bitmap Image	7/31/2001 1:29 PM

Fig. 13.7 `LinkLabel`s used to link to a folder, a Web page and an application. (Part 2 of 3.)


LinkLabelTest
Click to browse C:\
Click to visit www.deitel.com
Click to run Notepad
Deitel & Associates, Inc. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address
http://www.deitel.com/
Go
Links
DEITEL & ASSOCIATES INC.
Corporate Training Curriculum
Publications/Book Store
What's New/Upcoming Publications
Downloads/Resources
Book Errata
FAQ (Frequently Asked Questions)
Job Opportunities
Internships
Teaching Opportunities
Internet
LinkLabelTest
Click to browse C:\
Click to visit www.deitel.com
Click to run Notepad
Untitled - Notepad
File Edit Format Help

Java™, C, C++, Visual Basic®, Object Technology, and Internet and World Wide Web Prograr
On-Site Seminars Delivered Worldwide (Contact Us for on-site seminars on any programming to

deitel@deitel.com
978.579.9911
490B Boston Post Road, Suite 200, Sudbury, MA 01776

Full-time Job Opportunities

Available Immediately for Computer Science and/o
Technology Graduates at our Sudbury, MA lo

Deitel & Associates, Inc. has several highly competitive **full-time positions**

All positions require employees to live in the Boston

Fig. 13.7 `LinkLabel`s used to link to a folder, a Web page and an application. (Part 3 of 3.)

13.4 `ListBox`es and `CheckedListBox`es

The ***`ListBox`*** control allows the user to view and select from multiple items in a list. `ListBox`es are static GUI entities, which means that users cannot enter new items in the list. The ***`CheckedListBox`*** control extends a `ListBox` by including check boxes next to each item in the list. This allows users to place checks on multiple items at once, as is possible in a `CheckBox` control (users also can select multiple items simultaneously from a `ListBox`, but not by default). Figure 13.8 displays a sample `ListBox` and a sample `CheckedListBox`. In both controls, scroll bars appear if the number of items is too large to be displayed simultaneously in the component. Figure 13.9 lists common `ListBox` properties, methods and events.

The ***`SelectionMode`*** property determines the number of items that can be selected. This property has the possible values ***`None`***, ***`One`***, ***`MultiSimple`*** and ***`MultiExtended`*** (from the ***`SelectionMode`*** *enumeration*)—the differences among these settings are explained in Fig. 13.9. The `SelectedIndexChanged` event occurs when the user selects a new item.

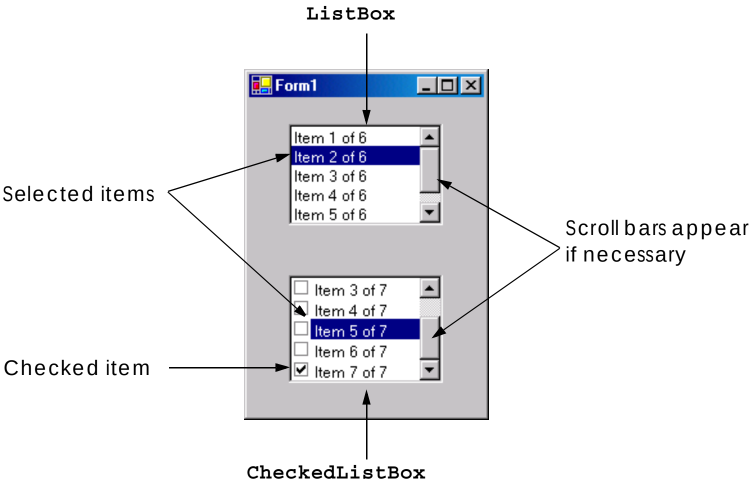

ListBox
Form1
Item 1 of 6
Item 2 of 6
Item 3 of 6
Item 4 of 6
Item 5 of 6
Selected items
Scroll bars appear if necessary
Item 3 of 7
Item 4 of 7
Item 5 of 7
Item 6 of 7
Item 7 of 7
Checked item
CheckedListBox

Fig. 13.8 `ListBox` and `CheckedListBox` on a form.

`ListBox` properties, methods and events	**Description / Delegate and Event Arguments**
Common Properties	
`Items`	Lists the collection of items within the **`ListBox`**.
`MultiColumn`	Indicates whether the **`ListBox`** can break a list into multiple columns. Multiple columns are used to make vertical scroll bars unnecessary.
`SelectedIndex`	Returns the index of the currently selected item. If the user selects multiple items, this method arbitrarily returns one of the selected indices; if no items have been selected, the method returns **`-1`**.
`SelectedIndices`	Returns a collection of the indices of all currently selected items.
`SelectedItem`	Returns a reference to the currently selected item (if multiple items are selected, it returns the item with the lowest index number).
`SelectedItems`	Returns a collection of the currently selected item(s).
`SelectionMode`	Determines the number of items that can be selected and the means through which multiple items can be selected. Values **`None`**, **`One`**, **`MultiSimple`** (multiple selection allowed) and **`MultiExtended`** (multiple selection allowed via a combination of arrow keys, mouse clicks and *Shift* and *Control* buttons).
`Sorted`	Indicates whether items appear in alphabetical order. **`True`** causes alphabetization; default is **`False`**.

Fig. 13.9 `ListBox` properties, methods and events. (Part 1 of 2.)

`ListBox` properties, methods and events	**Description / Delegate and Event Arguments**
Common Method	
`GetSelected`	Takes an index, and returns **`True`** if the corresponding item is selected.
Common Event	*(Delegate **`EventHandler`**, event arguments **`EventArgs`**)*
`SelectedIndexChanged`	Generated when selected index changes. Default when control is double-clicked in designer.

Fig. 13.9 **`ListBox`** properties, methods and events. (Part 2 of 2.)

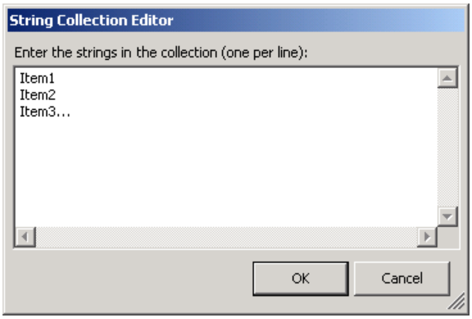
Both the **`ListBox`** and **`CheckedListBox`** have properties **`Items`**, **`SelectedItem`** and **`SelectedIndex`**. Property **`Items`** returns all the objects in the list as a collection. Collections are a common way of exposing lists of **`Object`**s in the .NET framework. Many .NET GUI components (e.g., **`ListBox`**es) use collections to expose lists of internal objects (e.g., items contained within a **`ListBox`**). We discuss collections further in Chapter 23, Data Structures and Collections. Property **`SelectedItem`** returns the currently selected item. If the user can select multiple items, use collection **`SelectedItems`** to return all the selected items as a collection. Property **`SelectedIndex`** returns the index of the selected item—if there could be more than one, use property **`SelectedIndices`**. If no items are selected, property **`SelectedIndex`** returns **`-1`**. Method **`GetSelected`** takes an index and returns **`True`** if the corresponding item is selected.

To add items to the **`ListBox`** or the **`CheckedListBox`** we must add objects to its **`Items`** collection. This can be accomplished by invoking method **`Add`** to add a **`String`** to the **`ListBox`**'s or **`CheckedListBox`**'s **`Items`** collection. For example, we could write

myListBox`.Items.Add(` *"myListItem"* `)`

to add **`String`** *myListItem* to **`ListBox`** *myListBox*. To add multiple objects, programmers can either use method **`Add`** multiple times or use method **`AddRange`** to add an array of objects. Classes **`ListBox`** and **`CheckedListBox`** use each submitted object's **`ToString`** method to determine the label for the corresponding object's entry in the list. This allows developers to add different objects to a **`ListBox`** or a **`CheckedListBox`** that later can be returned through properties **`SelectedItem`** and **`SelectedItems`**.

Alternatively, we can add items to **`ListBox`**es and **`CheckedListBox`**es visually by examining the **`Items`** property in the **Properties** window. Clicking the ellipsis opens the **String Collection Editor**, a text area in which we can type the items to add; each item should appear on a separate line (Fig. 13.10). Visual Studio .NET then adds these **`String`**s to the **`Items`** collection inside method **`InitializeComponent`**.



Enter the strings in the collection (one per line):

```
Item1
Item2
Item3...
```

Fig. 13.10 String Collection Editor.

13.4.1 `ListBox`es

Class **`ListBoxTest`** (Fig. 13.11) enables the user to add, remove and clear items from **`ListBox`** **`displayListBox`** (line 14). Class **`ListBoxTest`** uses **`TextBox`** **`inputTextBox`** (line 17) to allow the user to type in a new item. When the user clicks button **`addButton`** (line 20), the new item appears in **`displayListBox`**. Similarly, if the user selects an item and clicks **`removeButton`** (line 21), the item is deleted. Control **`clearButton`** (line 22) deletes all entries in **`displayListBox`**. The user terminates the application by clicking button **`exitButton`** (line 23).

The **`addButton_Click`** event handler (lines 33–38) calls method **`Add`** of the **`Items`** collection in the **`ListBox`**. This method takes a **`String`** as the item to add to **`displayListBox`**. In this case, the **`String`** used is the user-input text, or **`inputTextBox.Text`** (line 36). After the item is added, **`txtInput.Text`** is cleared (line 37).

The **`removeButton_Click`** event handler (lines 41–48) calls method **`Remove`** of the **`Items`** collection. Event handler **`removeButton_Click`** first uses property **`SelectedIndex`** to check which index is selected. Unless **`SelectedIndex`** is **`-1`** (line 45), the handler removes the item that corresponds to the selected index.

```
 1  // Fig 13.11: ListBoxTest.cs
 2  // Program to add, remove and clear list box items.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  public class ListBoxTest : System.Windows.Forms.Form
12  {
13     // contains user-input list of elements
14     private System.Windows.Forms.ListBox displayListBox;
15
```

Fig. 13.11 **`ListBox`** used in a program to add, remove and clear items. (Part 1 of 3.)

```
      // user input textbox
      private System.Windows.Forms.TextBox inputTextBox;

      // add, remove, clear and exit command buttons
      private System.Windows.Forms.Button addButton;
      private System.Windows.Forms.Button removeButton;
      private System.Windows.Forms.Button clearButton;
      private System.Windows.Forms.Button exitButton;

      [STAThread]
      static void Main()
      {
         Application.Run( new ListBoxTest() );
      }

      // add new item (text from input box)
      // and clear input box
      private void addButton_Click(
         object sender, System.EventArgs e )
      {
         displayListBox.Items.Add( inputTextBox.Text );
         inputTextBox.Clear();
      }

      // remove item if one selected
      private void removeButton_Click(
         object sender, System.EventArgs e )
      {
         // remove only if item selected
         if ( displayListBox.SelectedIndex != -1 )
            displayListBox.Items.RemoveAt(
               displayListBox.SelectedIndex );
      }

      // clear all items
      private void clearButton_Click(
         object sender, System.EventArgs e )
      {
         displayListBox.Items.Clear();
      }

      // exit application
      private void exitButton_Click(
         object sender, System.EventArgs e )
      {
         Application.Exit();
      }

   } // end class ListBoxTest
```

Fig. 13.11 `ListBox` used in a program to add, remove and clear items. (Part 2 of 3.)

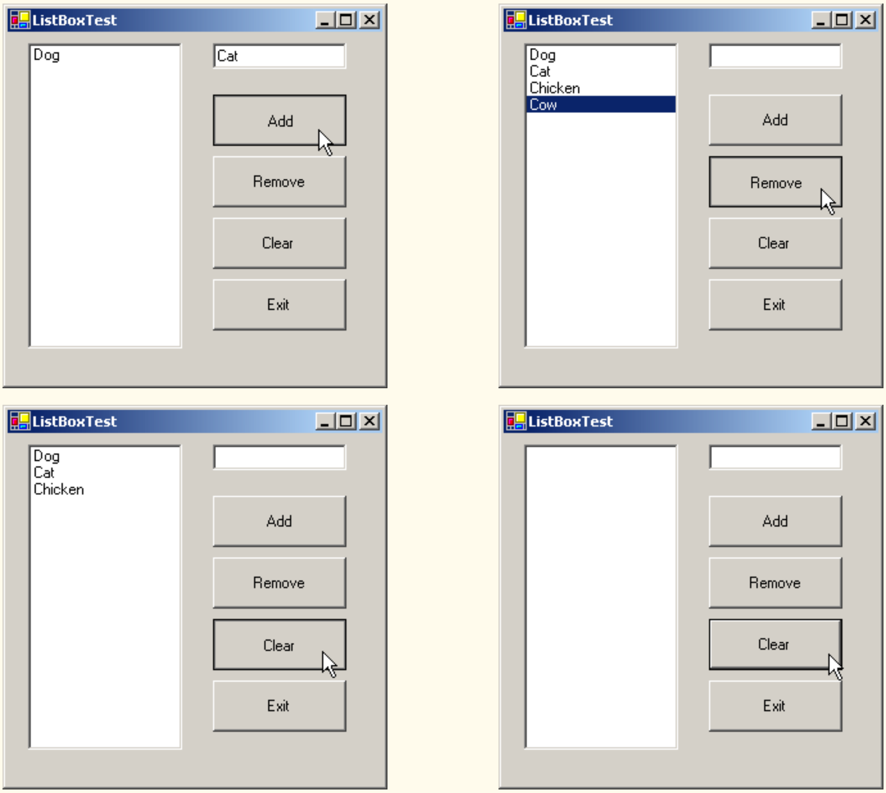

ListBoxTest
Dog
Cat
Add
Remove
Clear
Exit
ListBoxTest
Dog
Cat
Chicken
Cow
Add
Remove
Clear
Exit
ListBoxTest
Dog
Cat
Chicken
Add
Remove
Clear
Exit
ListBoxTest
Add
Remove
Clear
Exit

Fig. 13.11 `ListBox` used in a program to add, remove and clear items. (Part 3 of 3.)

The event handler for **`clearButton_Click`** (lines 51–55) calls method **`Clear`** of the **`Items`** collection (line 54). This removes all the entries in **`displayListBox`**. Finally, event handler **`exitButton_Click`** (lines 58–62) terminates the application, using method **`Application.Exit`** (line 61).

13.4.2 `CheckedListBox`es

The **`CheckedListBox`** control derives from class **`ListBox`** and includes a checkbox next to each item in the list. As in **`ListBox`**es, items can be added via methods **`Add`** and **`AddRange`** or through the **String Collection Editor**. **`CheckedListBox`**es imply that multiple items can be selected, and the only possible values for the **`SelectionMode`** property are **`SelectionMode.None`** and **`SelectionMode.One`**. **`SelectionMode.One`** allows multiple selection, because checkboxes imply that there are no logical restrictions on the items—the user can select as many items as required. Thus, the only choice is whether to give the user multiple selection or no selection at all. This keeps the **`CheckedListBox`**'s behavior consistent with that of **`CheckBox`**es. The programmer is unable to set the last two **`SelectionMode`** values, **`MultiSimple`** and **`MultiExtended`**, because the only logical selection modes are handled by **`None`** and **`One`**. Common properties and events of **`CheckedListBox`**es appear in Fig. 13.12.

Common Programming Error 13.1

The IDE displays an error message if the programmer attempts to set the `SelectionMode` property to `MultiSimple` or `MultiExtended` in the **Properties** *window of a `CheckedListBox`; if this value is set by the programmer in the code, a runtime error occurs.*

Event `ItemCheck` is generated whenever a user checks or unchecks a `CheckedListBox` item. Event argument properties `CurrentValue` and `NewValue` return `CheckState` values for the current and the new state of the item, respectively. A comparison of these values allows us to determine whether the `CheckedListBox` item was checked or unchecked. The `CheckedListBox` control retains the `SelectedItems` and `SelectedIndices` properties (it inherits them from class `ListBox`). However, it also includes properties `CheckedItems` and `CheckedIndices`, which return information about the checked items and indices.

In Fig. 13.13, class `CheckedListBoxTest` uses a `CheckedListBox` and a `ListBox` to display a user's selection of books. The `CheckedListBox` named `inputCheckedListBox` (lines 14–15) allows the user to select multiple titles. In the **String Collection Editor**, items were added for some Deitel books: C++, Java, VB, Internet & WWW, Perl, Python, Wireless Internet and Advanced Java (the acronym HTP stands for "How to Program"). The `ListBox`, named `displayListBox` (line 18), displays the user's selection. In the screen shots accompanying this example, the `CheckedListBox` appears to the left, the `ListBox` to the right.

`CheckedListBox` properties, methods and events	Description / Delegate and Event Arguments
Common Properties	*(All the `ListBox` properties and events are inherited by `CheckedListBox`.)*
`CheckedItems`	Lists the collection of items that are checked. This is distinct from the selected items, which are highlighted (but not necessarily checked). *Note: There can be at most one selected item at any given time.*
`CheckedIndices`	Returns indices for the items that are checked. Not the same as the selected indices.
`SelectionMode`	Determines how many items can be checked. Only possible values are `One` (allows multiple checks to be placed) or `None` (does not allow any checks to be placed).
Common Method	
`GetItemChecked`	Takes an index, and returns `True` if corresponding item is checked.
Common Event	*(Delegate `ItemCheckEventHandler`, event arguments `ItemCheckEventArgs`)*
`ItemCheck`	Generated when an item is checked or unchecked.

Fig. 13.12 `CheckedListBox` properties, methods and events. (Part 1 of 2.)

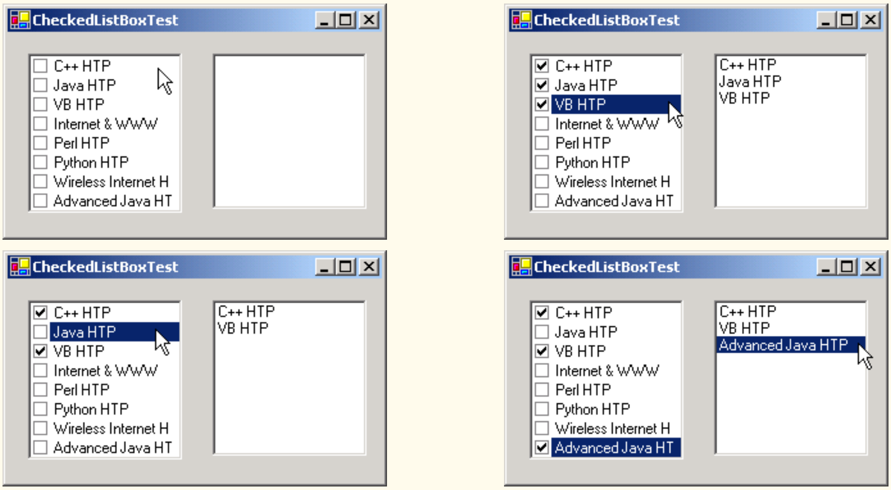
`CheckedListBox` **properties, methods and events**	**Description / Delegate and Event Arguments**
`ItemCheckEventArgs` *Properties*	
`CurrentValue`	Indicates whether current item is checked or unchecked. Possible values are **`Checked`**, **`Unchecked`** and **`Indeterminate`**.
`Index`	Returns index of the item that changed.
`NewValue`	Specifies the new state of item.

Fig. 13.12 **`CheckedListBox`** properties, methods and events. (Part 2 of 2.)

```
// Fig. 13.13: CheckedListBoxTest.cs
// Using the checked list boxes to add items to a list box

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class CheckedListBoxTest : System.Windows.Forms.Form
{
   // list of available book titles
   private System.Windows.Forms.CheckedListBox
      inputCheckedListBox;

   // user selection list
   private System.Windows.Forms.ListBox displayListBox;

   [STAThread]
   static void Main()
   {
      Application.Run( new CheckedListBoxTest() );
   }

   // item about to change,
   // add or remove from displayListBox
   private void inputCheckedListBox_ItemCheck(
      object sender,
      System.Windows.Forms.ItemCheckEventArgs e )
   {
      // obtain reference of selected item
      string item =
         inputCheckedListBox.SelectedItem.ToString();

```

Fig. 13.13 **`CheckedListBox`** and **`ListBox`** used in a program to display a user selection. (Part 1 of 2.)

```
36          // if item checked add to listbox
37          // otherwise remove from listbox
38          if ( e.NewValue == CheckState.Checked )
39             displayListBox.Items.Add( item );
40          else
41             displayListBox.Items.Remove( item );
42
43       } // end method inputCheckedListBox_Click
44
45    } // end class CheckedListBox
```



☐ C++ HTP
☐ Java HTP
☐ VB HTP
☐ Internet & WWW
☐ Perl HTP
☐ Python HTP
☐ Wireless Internet H
☐ Advanced Java HT

☑ C++ HTP
☑ Java HTP
☑ VB HTP
☐ Internet & WWW
☐ Perl HTP
☐ Python HTP
☐ Wireless Internet H
☐ Advanced Java HT

C++ HTP
Java HTP
VB HTP

☑ C++ HTP
☐ Java HTP
☑ VB HTP
☐ Internet & WWW
☐ Perl HTP
☐ Python HTP
☐ Wireless Internet H
☐ Advanced Java HT

C++ HTP
VB HTP

☑ C++ HTP
☐ Java HTP
☑ VB HTP
☐ Internet & WWW
☐ Perl HTP
☐ Python HTP
☐ Wireless Internet H
☑ Advanced Java HT

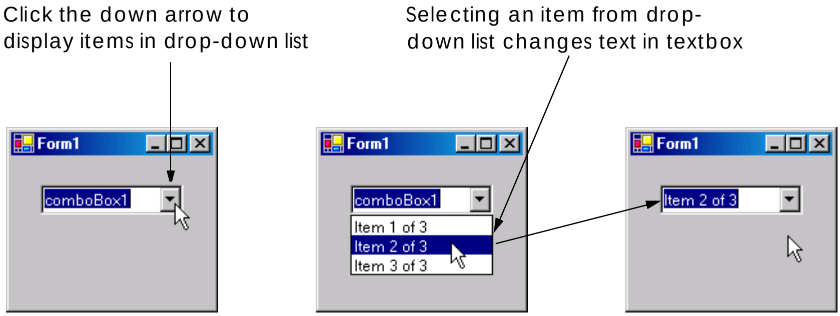
C++ HTP
VB HTP
Advanced Java HTP

Fig. 13.13 `CheckedListBox` and `ListBox` used in a program to display a user selection. (Part 2 of 2.)

When the user checks or unchecks an item in `CheckedListBox` `inputCheckedListBox`, the system generates an `ItemCheck` event. Event handler `inputCheckedListBox_ItemCheck` (lines 28–43) handles the event. An `if`/`else` control structure (lines 38–41) determines whether the user checked or unchecked an item in the `CheckedListBox`. Line 38 uses the `NewValue` property to test for whether the item is being checked (`CheckState.Checked`). If the user checks an item, line 39 adds the checked entry to the `ListBox` `displayListBox`. If the user unchecks an item, line 41 removes the corresponding item from `displayListBox`.

13.5 ComboBoxes

The ***`ComboBox`*** control combines `TextBox` features with a *drop-down list*. A drop-down list is a GUI component that contains a list from which values can be chosen. It usually appears as a text box with a down arrow to its right. By default, the user can enter text into the text box or click the down arrow to display a list of predefined items. If a user chooses an element from this list, that element is displayed in the text box. If the list contains more elements than can be displayed in the drop-down list, a scrollbar appears. The maximum number of items that a drop-down list can display at one time is set by property ***`MaxDropDownItems`***. Figure 13.14 shows a sample `ComboBox` in three different states.



Click the down arrow to display items in drop-down list

Selecting an item from drop-down list changes text in textbox

Fig. 13.14 **`ComboBox`** demonstration.

As with the **`ListBox`** control, the developer can add objects to collection **`Items`** programmatically, using methods **`Add`** and **`AddRange`**, or visually, with the **String Collection Editor**. Figure 13.15 lists common properties and events of class **`ComboBox`**.

`ComboBox` events and properties	**Description / Delegate and Event Arguments**
Common Properties	
`DropDownStyle`	Determines the type of combo box. Value **`Simple`** means that the text portion is editable and the list portion is always visible. Value **`DropDown`** (the default) means that the text portion is editable, but the user must click an arrow button to see the list portion. Value **`DropDownList`** means that the text portion is not editable and the user must click the arrow button to see the list portion.
`Items`	The collection of items in the **`ComboBox`** control.
`MaxDropDownItems`	Specifies the maximum number of items (between **`1`** and **`100`**) that can display in the drop-down list. If the number of items exceeds the maximum number of items to display, a scroll bar appears.
`SelectedIndex`	Returns index of currently selected item. If there is no currently selected item, **`-1`** is returned.
`SelectedItem`	Returns a reference to the currently selected item.
`Sorted`	Specifies whether items in a list are alphabetized. If **`True`**, items appear in alphabetical order. Default is **`False`**.
Common Event	*(Delegate **`EventHandler`**, event arguments **`EventArgs`**)*
`SelectedIndexChanged`	Generated when the selected index changes (such as when a check box has been checked or unchecked). Default when control is double-clicked in designer.

Fig. 13.15 **`ComboBox`** properties and events.

Look-and-Feel Observation 13.6

Use a `ComboBox` to save space on a GUI. The disadvantage is that, unlike with a `ListBox`, the user cannot see available items without scrolling.

Property *`DropDownStyle`* determines the type of `ComboBox`. Style *`Simple`* does not display a drop-down arrow. Instead, a scrollbar appears next to the control, allowing the user to select a choice from the list. The user can also type in a selection. Style *`DropDown`* (the default) displays a drop-down list when the down arrow is clicked (or the down arrow key is pressed). The user can type a new item into the `ComboBox`.

The last style is *`DropDownList`*, which displays a drop-down list but does not allow the user to enter a new item. Drop-down lists save room, so a `ComboBox` should be used when GUI space is limited.

The `ComboBox` control has properties `Items` (a collection), `SelectedItem` and `SelectedIndex`, which are similar to the corresponding properties in `ListBox`. There can be at most one selected item in a `ComboBox` (if zero, then `SelectedIndex` is `-1`). When the selected item changes, event `SelectedIndexChanged` is generated.

Class `ComboBoxTest` (Fig. 13.16) allows users to select a shape to draw—an empty or filled circle, ellipse, square or pie—by using a `ComboBox`. The combo box in this example is uneditable, so the user cannot input a custom item.

Look-and-Feel Observation 13.7

Make lists (such as `ComboBox`es) editable only if the program is designed to accept user-submitted elements. Otherwise, the user might enter a custom item and then be unable to use it.

After creating `ComboBox imageComboBox` (line 14), we make it uneditable by setting its `DropDownStyle` to `DropDownList` in the **Properties** window. Next, we add items `Circle`, `Square`, `Ellipse`, `Pie`, `Filled Circle`, `Filled Square`, `Filled Ellipse` and `Filled Pie` to the `Items` collection. We added these items using the **String Collection Editor**. Whenever the user selects an item from `imageComboBox`, the system generates a `SelectedIndexChanged` event. Event handler `imageComboBox_SelectedIndexChanged` (lines 23–77) handles these events. Lines 27–34 create a `Graphics` object, a `Pen` and a `SolidBrush`, with which the program draws on the form. The `Graphics` object (line 22) allows a pen or brush to draw on a component, using one of several `Graphics` methods. The `Pen` object is used by methods `drawEllipse`, `drawRectangle` and `drawPie` (lines 43–56) to draw the outlines of their corresponding shapes. The `SolidBrush` object is used by methods `fillEllipse`, `fillRectangle` and `fillPie` (lines 59–72) to draw their corresponding solid shapes. Line 37 colors the entire form `White`, using `Graphics` method `Clear`. These methods are discussed in greater detail in Chapter 16, Graphics and Multimedia.

```
1  // Fig. 13.16: ComboBoxTest.cs
2  // Using ComboBox to select shape to draw
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
```

Fig. 13.16 `ComboBox` used to draw a selected shape. (Part 1 of 3.)

```
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class ComboBoxTest : System.Windows.Forms.Form
{
   // contains shape list (circle, square, ellipse, pie)
   private System.Windows.Forms.ComboBox imageComboBox;

   [STAThread]
   static void Main()
   {
      Application.Run( new ComboBoxTest() );
   }

   // get selected index, draw shape
   private void imageComboBox_SelectedIndexChanged(
      object sender, System.EventArgs e )
   {
      // create graphics object, pen and brush
      Graphics myGraphics = base.CreateGraphics();

      // create Pen using color DarkRed
      Pen myPen = new Pen( Color.DarkRed );

      // create SolidBrush using color DarkRed
      SolidBrush mySolidBrush =
         new SolidBrush( Color.DarkRed );

      // clear drawing area setting it to color White
      myGraphics.Clear( Color.White );

      // find index, draw proper shape
      switch ( imageComboBox.SelectedIndex )
      {
         case 0: // case circle is selected
            myGraphics.DrawEllipse(
               myPen, 50, 50, 150, 150 );
            break;
         case 1: // case rectangle is selected
            myGraphics.DrawRectangle(
               myPen, 50, 50, 150, 150 );
            break;
         case 2: // case ellipse is selected
            myGraphics.DrawEllipse(
               myPen, 50, 85, 150, 115 );
            break;
         case 3: // case pie is selected
            myGraphics.DrawPie(
               myPen, 50, 50, 150, 150, 0, 45 );
            break;
```

Fig. 13.16 `ComboBox` used to draw a selected shape. (Part 2 of 3.)

```
         case 4: // case filled circle is selected
            myGraphics.FillEllipse(
               mySolidBrush, 50, 50, 150, 150 );
            break;
         case 5: // case filled rectangle is selected
            myGraphics.FillRectangle(
               mySolidBrush, 50, 50, 150, 150 );
            break;
         case 6: // case filled ellipse is selected
            myGraphics.FillEllipse(
               mySolidBrush, 50, 85, 150, 115 );
            break;
         case 7: // case filled pie is selected
            myGraphics.FillPie(
               mySolidBrush, 50, 50, 150, 150, 0, 45 );
            break;

      } // end switch

   } // end method imageComboBox_SelectedIndexChanged

} // end class ComboBoxTest
```

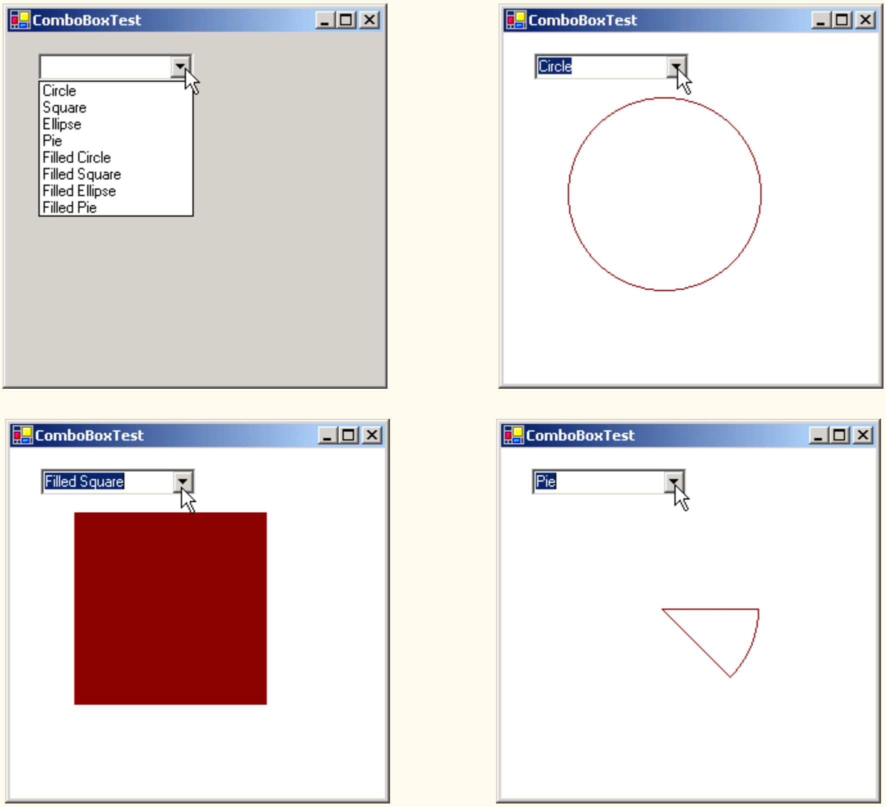

ComboBoxTest
Circle
Square
Ellipse
Pie
Filled Circle
Filled Square
Filled Ellipse
Filled Pie
ComboBoxTest
Circle
ComboBoxTest
Filled Square
ComboBoxTest
Pie

Fig. 13.16 `ComboBox` used to draw a selected shape. (Part 3 of 3.)

The application draws a particular shape specified by the selected item's index. The **`switch`** structure (lines 40–75) uses **`imageComboBox.SelectedIndex`** to determine which item the user selected. Class **`Graphics`** method ***`DrawEllipse`*** (lines 43–44) takes a **`Pen`**, the *x*- and *y*-coordinates of the center and the width and height of the ellipse to draw. The origin of the coordinate system is in the upper left corner of the form; the *x*-coordinate increases to the right, the *y*-coordinate increases downward. A circle is a special case of an ellipse (the height and width are equal). Lines 43–44 draw a circle. Lines 51–52 draw an ellipse that has different values for height and width.

Class **`Graphics`** method ***`DrawRectangle`*** (lines 47–48) takes a **`Pen`**, the *x*- and *y*-coordinates of the upper-left corner and the width and height of the rectangle to draw. Method ***`DrawPie`*** (line 55–56) draws a pie as a portion of an ellipse. The ellipse is bounded by a rectangle. Method **`DrawPie`** takes a **`Pen`**, the *x*- and *y*-coordinates of the upper-left corner of the rectangle, its width and height, the start angle (in degrees) and the sweep angle (in degrees) of the pie. Angles increase clockwise. The ***`FillEllipse`*** (lines 59–60 and 67–68), ***`FillRectange`*** (lines 63–64) and ***`FillPie`*** (lines 71–72) methods are similar to their unfilled counterparts, except that they take a **`SolidBrush`** instead of a **`Pen`**. Some of the drawn shapes are illustrated in the screen shots at the bottom of Fig. 13.16.

13.6 `TreeViews`

The ***`TreeView`*** control displays *nodes* hierarchically on a *tree*. Traditionally, nodes are objects that contain values and can refer to other nodes. A *parent node* contains *child nodes*, and the child nodes can be parents to other nodes. Two child nodes that have the same parent node are considered *sibling nodes*. A tree is a collection of nodes, usually organized in a hierarchical manner. The first parent node of a tree is the *root* node (a **`TreeView`** can have multiple roots). For example, the file system of a computer can be represented as a tree. The top-level directory (perhaps **`C:`**) would be the root, each subfolder of **`C:`** would be a child node and each child folder could have its own children. **`TreeView`** controls are useful for displaying hierarchal information, such as the file structure that we just mentioned. We cover nodes and trees in greater detail in Chapter 24, Data Structures. Figure 13.17 displays a sample **`TreeView`** control on a form.

A parent node can be expanded or collapsed by clicking the plus or minus box to its left. Nodes without children do not have an expand or collapse box.

The nodes displayed in a **`TreeView`** are instances of class **`TreeNode`**. Each **`TreeNode`** has a **`Nodes`** collection (type **`TreeNodeCollection`**), which contains a list of other **`TreeNode`**s—its children. The **`Parent`** property returns a reference to the parent node (or **`null`** if the node is a root node). Figure 13.18 and Fig. 13.19 list the common properties of **`TreeView`**s and **`TreeNode`**s and an event of **`TreeView`**s.

To add nodes to the **`TreeView`** visually, click the ellipsis by the **`Nodes`** property in the **Properties** window. This opens the **TreeNode Editor**, which displays an empty tree representing the **`TreeView`** (Fig. 13.20). There are buttons to create a root, to add or delete a node.

To add nodes through code, we first must create a root node. Make a new **`TreeNode`** object and pass it a **`String`** to display. Then, use method **`Add`** to add this new **`TreeNode`** to the **`TreeView`**'s **`Nodes`** collection. Thus, to add a root node to **`TreeView`** *myTreeView*, write *myTreeView*`.Nodes.Add(New TreeNode(`*RootLabel*`))`

where *myTreeView* is the **`TreeView`** to which we are adding nodes, and *RootLabel* is the text to display in *myTreeView*. To add children to a root node, add new **`TreeNode`**s to its **`Nodes`** collection. We select the appropriate root node from the **`TreeView`** by writing

myTreeView`.Nodes(`*myIndex*`)`

where *myIndex* is the root node's index in *myTreeView*'s **`Nodes`** collection. We add nodes to child nodes through the same process by which we added root nodes to *myTreeView*. To add a child to the root node at index *myIndex*, write

myTreeView`.Nodes(`*myIndex*`).Nodes.Add(New TreeNode(`*ChildLabel*`))`

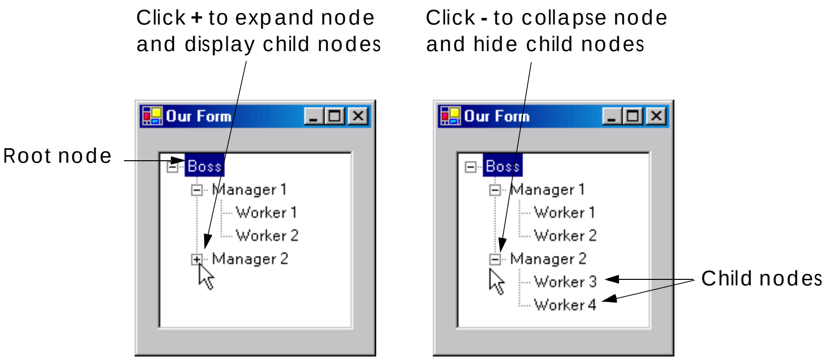

Click + to expand node and display child nodes
Click - to collapse node and hide child nodes
Root node
Child nodes
Our Form
Boss
Manager 1
Worker 1
Worker 2
Manager 2
Worker 3
Worker 4

Fig. 13.17 **`TreeView`** displaying a sample tree.

`TreeView` properties and events	**Description / Delegate and Event Arguments**
Common Properties	
`CheckBoxes`	Indicates whether checkboxes appear next to nodes. **`True`** displays checkboxes. Default is **`False`**.
`ImageList`	Indicates the **`ImageList`** used to display icons by the nodes. An ***`ImageList`*** is a collection that contains a number of **`Image`** objects.
`Nodes`	Lists the collection of **`TreeNode`**s in the control. Contains methods **`Add`** (adds a **`TreeNode`** object), **`Clear`** (deletes the entire collection) and **`Remove`** (deletes a specific node). Removing a parent node deletes all its children.

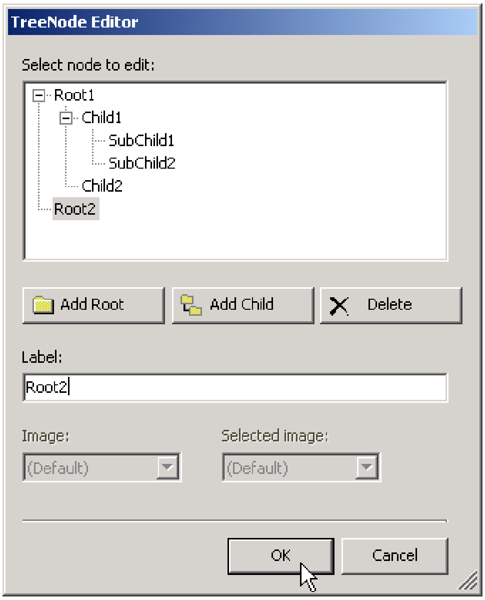
Fig. 13.18 **`TreeView`** properties and events. (Part 1 of 2.)

`TreeView` **properties and events**	**Description / Delegate and Event Arguments**
`SelectedNode`	Currently selected node.
Common Event	*(Delegate* **`TreeViewEventHandler`**, *event arguments* **`TreeViewEventArgs`***)*
`AfterSelect`	Generated after selected node changes. Default when double-clicked in designer.

Fig. 13.18 **`TreeView`** properties and events. (Part 2 of 2.)

`TreeNode` **properties and methods**	**Description / Delegate and Event Arguments**
Common Properties	
`Checked`	Indicates whether the **`TreeNode`** is checked. (**`CheckBoxes`** property must be set to **`True`** in parent **`TreeView`**.)
`FirstNode`	Specifies the first node in the **`Nodes`** collection (i.e., first child in tree).
`FullPath`	Indicates the path of the node, starting at the root of the tree.
`ImageIndex`	Specifies the index of the image to be shown when the node is deselected.
`LastNode`	Specifies the last node in the **`Nodes`** collection (i.e., last child in tree).
`NextNode`	Next sibling node.
`Nodes`	The collection of **`TreeNode`**s contained in the current node (i.e., all the children of the current node). Contains methods **`Add`** (adds a **`TreeNode`** object), **`Clear`** (deletes the entire collection) and **`Remove`** (deletes a specific node). Removing a parent node deletes all its children.
`PrevNode`	Indicates the previous sibling node.
`SelectedImageIndex`	Specifies the index of the image to use when the node is selected.
`Text`	Specifies the text to display in the **`TreeView`**.
Common Methods	
`Collapse`	Collapses a node.
`Expand`	Expands a node.
`ExpandAll`	Expands all the children of a node.
`GetNodeCount`	Returns the number of child nodes.

Fig. 13.19 **`TreeNode`** properties and methods.



TreeNode Editor

Select node to edit:

- Root1
  - Child1
    - SubChild1
    - SubChild2
  - Child2
- Root2

Label: Root2

Image: (Default)

Selected image: (Default)

Fig. 13.20 TreeNode Editor.

Class `TreeViewDirectoryStructureTest` (Fig. 13.21) uses a `TreeView` to display the directory file structure on a computer. The root node is the `C:` drive, and each subfolder of `C:` becomes a child. This layout is similar to that used in **Windows Explorer**. Folders can be expanded or collapsed by clicking the plus or minus boxes that appear to their left.

When `TreeViewDirectoryStructureTest` loads, the system generates a `Load` event, which is handled by event handler `TreeViewDirectoryStructureTest_Load` (lines 64–72). Line 69 adds a root node (`C:`) to our `TreeView`, named `directoryTreeView`. `C:` is the root folder for the entire directory structure. Lines 70–71 call method `PopulateTreeView` (lines 25–61), which takes a directory (a `String`) and a parent node. Method `PopulateTreeView` then creates child nodes corresponding to the subdirectories of the directory that was passed to it.

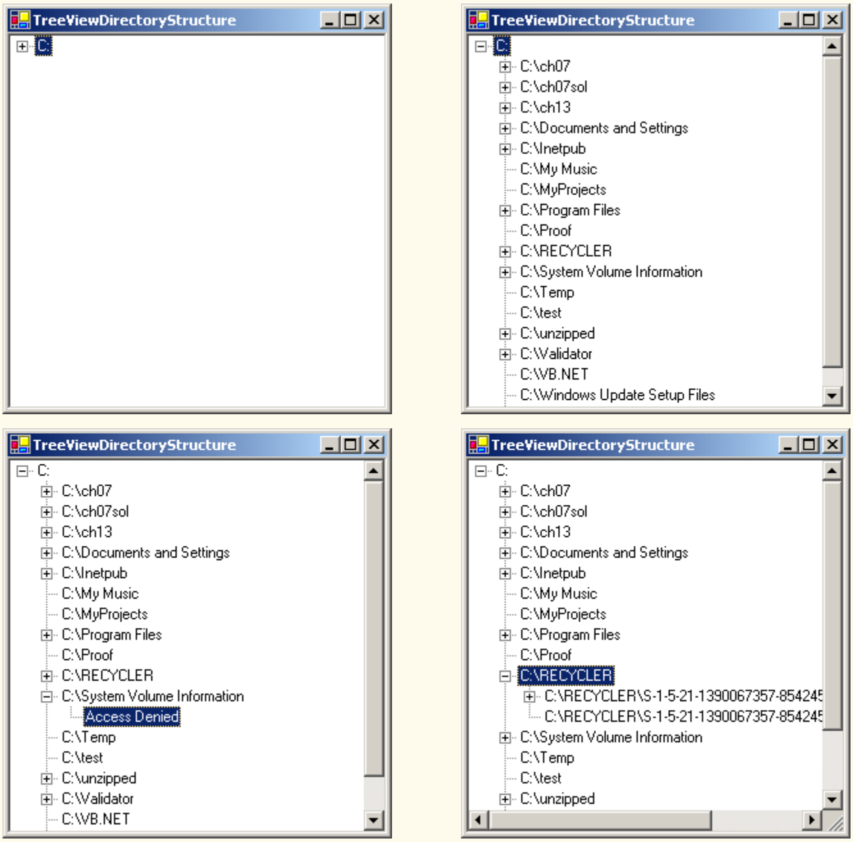
```
1  // Fig. 13.21: TreeViewDirectoryStructureTest.cs
2  // Using TreeView to display directory structure
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Data;
10 using System.IO;
11
```

Fig. 13.21 `TreeView` used to display directories. (Part 1 of 3.)

```
public class TreeViewDirectoryStructureTest
   : System.Windows.Forms.Form
{
   // contains view of c: drive directory structure
   private System.Windows.Forms.TreeView directoryTreeView;

   [STAThread]
   static void Main()
   {
      Application.Run(
         new TreeViewDirectoryStructureTest() );
   }

   public void PopulateTreeView(
      string directoryValue, TreeNode parentNode )
   {
      // populate current node with subdirectories
      string[] directoryArray =
         Directory.GetDirectories( directoryValue );

      // populate current node with subdirectories
      try
      {
         if ( directoryArray.Length != 0 )
         {
            // for every subdirectory, create new TreeNode,
            // add as child of current node and recursively
            // populate child nodes with subdirectories
            foreach ( string directory in directoryArray )
            {
               // create TreeNode for current directory
               TreeNode myNode = new TreeNode( directory );

               // add current directory node to parent node
               parentNode.Nodes.Add( myNode );

               // recursively populate every subdirectory
               PopulateTreeView( directory, myNode );
            }

         } // end if
      }

      // catch exception
      catch ( UnauthorizedAccessException )
      {
         parentNode.Nodes.Add( "Access denied" );
      }

   } // end PopulateTreeView

```

Fig. 13.21 `TreeView` used to display directories. (Part 2 of 3.)

```
63     // called by system when form loads
64     private void TreeViewDirectoryStructureTest_Load(
65        object sender, System.EventArgs e)
66     {
67        // add c:\ drive to directoryTreeView and
68        // insert its subfolders
69        directoryTreeView.Nodes.Add( "C:\\" );
70        PopulateTreeView(
71           "C:\\", directoryTreeView.Nodes[ 0 ] );
72     }
73
74  } // end class TreeViewDirectoryStructure
```



```
TreeViewDirectoryStructure
⊞ C:
```

```
TreeViewDirectoryStructure
⊟ C:
   ⊞ C:\ch07
   ⊞ C:\ch07sol
   ⊞ C:\ch13
   ⊞ C:\Documents and Settings
   ⊞ C:\Inetpub
     C:\My Music
     C:\MyProjects
   ⊞ C:\Program Files
     C:\Proof
   ⊞ C:\RECYCLER
   ⊞ C:\System Volume Information
     C:\Temp
     C:\test
   ⊞ C:\unzipped
   ⊞ C:\Validator
     C:\VB.NET
     C:\Windows Update Setup Files
```

```
TreeViewDirectoryStructure
⊟ C:
   ⊞ C:\ch07
   ⊞ C:\ch07sol
   ⊞ C:\ch13
   ⊞ C:\Documents and Settings
   ⊞ C:\Inetpub
     C:\My Music
     C:\MyProjects
   ⊞ C:\Program Files
     C:\Proof
   ⊞ C:\RECYCLER
   ⊟ C:\System Volume Information
        Access Denied
     C:\Temp
     C:\test
   ⊞ C:\unzipped
   ⊞ C:\Validator
     C:\VB.NET
```

```
TreeViewDirectoryStructure
⊟ C:
   ⊞ C:\ch07
   ⊞ C:\ch07sol
   ⊞ C:\ch13
   ⊞ C:\Documents and Settings
   ⊞ C:\Inetpub
     C:\My Music
     C:\MyProjects
   ⊞ C:\Program Files
     C:\Proof
   ⊟ C:\RECYCLER
      ⊞ C:\RECYCLER\S-1-5-21-1390067357-854245
        C:\RECYCLER\S-1-5-21-1390067357-854245
   ⊞ C:\System Volume Information
     C:\Temp
     C:\test
   ⊞ C:\unzipped
```

Fig. 13.21 `TreeView` used to display directories. (Part 3 of 3.)

Method **`PopulateTreeView`** (lines 25–61) obtains a list of subdirectories, using method ***`GetDirectories`*** of class **`Directory`** (namespace **`System.IO`**) on lines 29–30. Method **`GetDirectories`** takes a **`String`** (the current directory) and returns an array of **`String`**s (the subdirectories). If a directory is not accessible for security reasons, an **`UnauthorizedAccessException`** is thrown. Lines 56–59 catch this exception and add a node containing “**`Access Denied`**” instead of displaying the subdirectories.

If there are accessible subdirectories, each **`String`** in the **`directoryArray`** is used to create a new child node (line 43). We use method **`Add`** (line 46) to add each child node to the parent. Then, method **`PopulateTreeView`** is called recursively on every subdirectory (line 49) and eventually populates the entire directory structure. Our recursive algorithm causes our program to have an initial delay when it loads—it must create a tree for the entire **`C:`** drive. However, once the drive folder names are added to the appropriate **`Nodes`** collection, they can be expanded and collapsed without delay. In the next section, we present an alternative algorithm to solve this problem.

13.7 `ListViews`

The ***`ListView`*** control is similar to a **`ListBox`**, in that both display lists from which the user can select one or more items (to see an example of a **`ListView`**, look ahead to the output of Fig. 13.24). The important difference between the two classes is that a **`ListView`** can display icons alongside the list items in a variety of ways (controlled by its **`ImageList`** property). Property ***`MultiSelect`*** (a boolean) determines whether multiple items can be selected. Checkboxes can be included by setting property **`CheckBoxes`** (a boolean) to **`True`**, making the **`ListView`**'s appearance similar to that of a **`CheckedListBox`**. The **`View`** property specifies the layout of the **`ListBox`**. Property ***`Activation`*** determines the method by which the user selects a list item. The details of these properties are explained in Fig. 13.22.

`ListView` allows us to define the images used as icons for **`ListView`** items. To display images, we must use an **`ImageList`** component. Create one by dragging it onto a form from the **ToolBox**. Then, click the **`Images`** collection in the **Properties** window to display the **Image Collection Editor** (Fig. 13.23). Here, developers can browse for images that they wish to add to the **`ImageList`**, which contains an array of **`Image`**s. Once the images have been defined, set property **`SmallImageList`** of the **`ListView`** to the new **`ImageList`** object. Property **`SmallImageList`** specifies the image list for the small icons. Property **`LargeImageList`** sets the **`ImageList`** for large icons. Icons for the **`ListView`** items are selected by setting the item's **`ImageIndex`** property to the appropriate array index.

`ListView` events and properties	**Description / Delegate and Event Arguments**
Common Properties	
`Activation`	Determines how the user activates an item. This property takes a value in the **`ItemActivation`** enumeration. Possible values are **`OneClick`** (single-click activation), **`TwoClick`** (double-click activation, item changes color when selected) and **`Standard`** (double-click activation).
`CheckBoxes`	Indicates whether items appear with checkboxes. **`True`** displays checkboxes. Default is **`False`**.
`LargeImageList`	Indicates the **`ImageList`** used when displaying large icons.

Fig. 13.22 **`ListView`** properties and events. (Part 1 of 2.)

`ListView` events and properties	**Description / Delegate and Event Arguments**
`Items`	Returns the collection of `ListViewItem`s in the control.
`MultiSelect`	Determines whether multiple selection is allowed. Default is `True`, which enables multiple selection.
`SelectedItems`	Lists the collection of currently selected items.
`SmallImageList`	Specifies the `ImageList` used when displaying small icons.
`View`	Determines appearance of `ListViewItem`s. Values `LargeIcon` (large icon displayed, items can be in multiple columns), `SmallIcon` (small icon displayed), `List` (small icons displayed, items appear in a single column) and `Details` (like `List`, but multiple columns of information can be displayed per item).
Common Event	*(Delegate* ***`EventHandler`****, event arguments* ***`EventArgs`****)*
`ItemActivate`	Generated when an item in the `ListView` is activated. Does not specify which item is activated.

Fig. 13.22 `ListView` properties and events. (Part 2 of 2.)

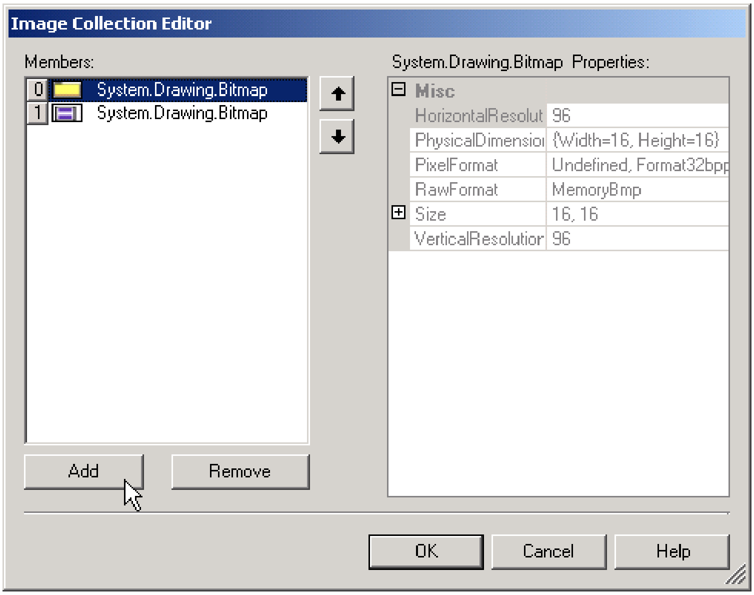


Image Collection Editor

Members:

0 System.Drawing.Bitmap
1 System.Drawing.Bitmap

System.Drawing.Bitmap Properties:

⊟ Misc

HorizontalResolut	96
PhysicalDimensioı	{Width=16, Height=16}
PixelFormat	Undefined, Format32bpp
RawFormat	MemoryBmp
⊞ Size	16, 16
VerticalResolutior	96

Fig. 13.23 **Image Collection Editor** window for an `ImageList` component.

Class `ListViewTest` (Fig. 13.24) displays files and folders in a `ListView`, along with small icons representing each file or folder. If a file or folder is inaccessible because of permission settings, a message box appears. The program scans the contents of the directory as it browses, rather than indexing the entire drive at once.

```
// Fig. 13.24: ListViewTest.cs
// Displaying directories and their contents in ListView.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

public class ListViewTest : System.Windows.Forms.Form
{
   // display labels for current location
   // in directory tree
   private System.Windows.Forms.Label currentLabel;
   private System.Windows.Forms.Label displayLabel;

   // display contents of current directory
   private System.Windows.Forms.ListView browserListView;

   // specifies images for file icons and folder icons
   private System.Windows.Forms.ImageList fileFolder;

   // get current directory
   string currentDirectory =
      Directory.GetCurrentDirectory();

   [STAThread]
   static void Main()
   {
      Application.Run( new ListViewTest() );
   }

   // browse directory user clicked or go up one level
   private void browserListView_Click(
      object sender, System.EventArgs e )
   {
      // ensure item selected
      if ( browserListView.SelectedItems.Count != 0 )
      {
         // if first item selected, go up one level
         if ( browserListView.Items[ 0 ].Selected )
         {
            // create DirectoryInfo object for directory
            DirectoryInfo directoryObject =
               new DirectoryInfo( currentDirectory );

            // if directory has parent, load it
            if ( directoryObject.Parent != null )
               LoadFilesInDirectory(
                  directoryObject.Parent.FullName );
         }
```

Fig. 13.24 `ListView` displaying files and folders. (Part 1 of 4.)

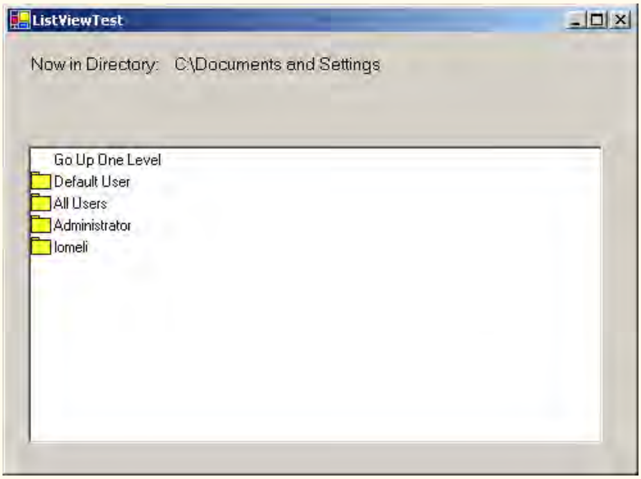
```

         // selected directory or file
         else
         {
            // directory or file chosen
            string chosen =
               browserListView.SelectedItems[ 0 ].Text;

            // if item selected is directory
            if ( Directory.Exists( currentDirectory +
               "\\" + chosen ) )
            {
               // load subdirectory
               // if in c:\, do not need '\',
               // otherwise we do
               if ( currentDirectory == "C:\\" )
                  LoadFilesInDirectory(
                     currentDirectory + chosen );
               else
                  LoadFilesInDirectory(
                     currentDirectory + "\\" + chosen );
            } //end if

         } // end else

         // update displayLabel
         displayLabel.Text = currentDirectory;

      } // end if

   } // end method browserListView_Click

   // display files/subdirectories of current directory
   public void LoadFilesInDirectory(
      string currentDirectoryValue )
   {
      // load directory information and display
      try
      {
         // clear ListView and set first item
         browserListView.Items.Clear();
         browserListView.Items.Add( "Go Up One Level" );

         // update current directory
         currentDirectory = currentDirectoryValue;
         DirectoryInfo newCurrentDirectory =
            new DirectoryInfo( currentDirectory );

         // put files and directories into arrays
         DirectoryInfo[] directoryArray =
            newCurrentDirectory.GetDirectories();

```

Fig. 13.24 `ListView` displaying files and folders. (Part 2 of 4.)

```
            FileInfo[] fileArray =
               newCurrentDirectory.GetFiles();

            // add directory names to ListView
            foreach ( DirectoryInfo dir in directoryArray )
            {
               // add directory to ListView
               ListViewItem newDirectoryItem =
                  browserListView.Items.Add( dir.Name );

               // set directory image
               newDirectoryItem.ImageIndex = 0;
            }

            // add file names to ListView
            foreach ( FileInfo file in fileArray )
            {
               // add file to ListView
               ListViewItem newFileItem =
                  browserListView.Items.Add( file.Name );

               newFileItem.ImageIndex = 1;  // set file image
            }
         } // end try

         // access denied
         catch ( UnauthorizedAccessException exception )
         {
            MessageBox.Show(
               "Warning: Some fields may not be " +
               "visible due to permission settings",
               "Attention", 0, MessageBoxIcon.Warning );
         }

      } // end method LoadFilesInDirectory

      // handle load event when Form displayed for first time
      private void ListViewTest_Load(
         object sender, System.EventArgs e )
      {
         // set image list
         Image folderImage = Image.FromFile(
            currentDirectory + "\\images\\folder.bmp" );

         Image fileImage = Image.FromFile( currentDirectory +
            "\\images\\file.bmp" );

         fileFolder.Images.Add( folderImage );
         fileFolder.Images.Add( fileImage );

         // load current directory into browserListView
         LoadFilesInDirectory( currentDirectory );
         displayLabel.Text = currentDirectory;
```

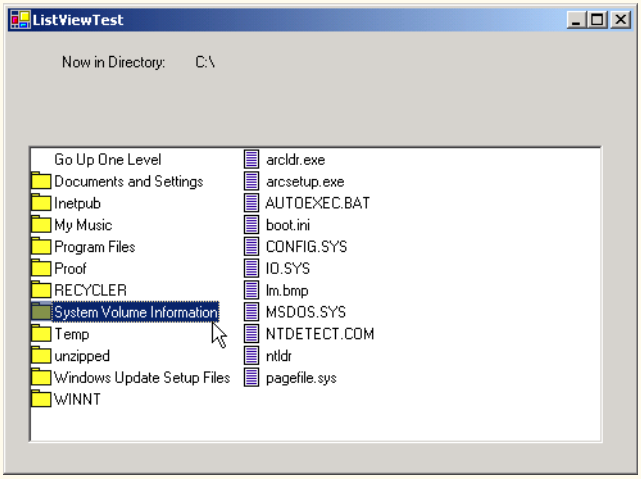
Fig. 13.24 `ListView` displaying files and folders. (Part 3 of 4.)

```
159
160      }  // end method ListViewTest_Load
161
162  } // end class ListViewTest
```



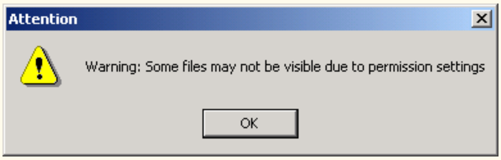
Now in Directory: C:\Documents and Settings

```
Go Up One Level
Default User
All Users
Administrator
lomeli
```



Now in Directory: C:\

```
Go Up One Level                  arcldr.exe
Documents and Settings           arcsetup.exe
Inetpub                          AUTOEXEC.BAT
My Music                         boot.ini
Program Files                    CONFIG.SYS
Proof                            IO.SYS
RECYCLER                         lm.bmp
System Volume Information        MSDOS.SYS
Temp                             NTDETECT.COM
unzipped                         ntldr
Windows Update Setup Files       pagefile.sys
WINNT
```



Warning: Some files may not be visible due to permission settings

Fig. 13.24 `ListView` displaying files and folders. (Part 4 of 4.)

To display icons beside list items, we must create an **`ImageList`** for the **`ListView`** **`browserListView`** (line 20). First, drag and drop an **`ImageList`** onto the form and open the **Image Collection Editor**. Create two simple bitmap images—one for a folder (array index 0) and another for a file (array index 1). Then, set the object **`browserListView`** property **`SmallImageList`** to the new **`ImageList`** in the **Properties** window. Developers can create such icons with any image software, such as Adobe® Photoshop™, Jasc® Paint Shop Pro™, or Microsoft® Paint.

Method **`LoadFilesInDirectory`** (lines 87–140) is used to populate **`browserListView`** with the directory passed to it (**`currentDirectoryValue`**). It clears **`browserListView`** and adds the element **`"Go Up One Level"`**. When the user clicks this element, the program attempts to move up one level (we see how shortly). The method then creates a **`DirectoryInfo`** object initialized with the string **`currentDirectory`** (lines 99–100). If permission is not given to browse the directory, an exception is thrown (caught on lines 132–138). Method **`LoadFilesInDirectory`** works differently from method **`PopulateTreeView`** in the previous program (Fig. 13.21). Instead of loading all the folders in the entire hard drive, method **`LoadFilesInDirectory`** loads only the folders in the current directory.

Class ***`DirectoryInfo`*** (namespace **`System.IO`**) enables us to browse or manipulate the directory structure easily. Method **`GetDirectories`** (lines 103–104) returns an array of **`DirectoryInfo`** objects containing the subdirectories of the current directory. Similarly, method **`GetFiles`** (lines 106–107) returns an array of class **`FileInfo`** objects containing the files in the current directory. Property ***`Name`*** (of both class **`DirectoryInfo`** and class **`FileInfo`**) contains only the directory or file name, such as **`temp`** instead of **`C:\myfolder\temp`**. To access the full name, use property ***`FullName`***.

Lines 110–118 and lines 121–128 iterate through the subdirectories and files of the current directory and add them to **`browserListView`**. Lines 117 and 127 set the **`ImageIndex`** properties of the newly created items. If an item is a directory, we set its icon to a directory icon (index 0); if an item is a file, we set its icon to a file icon (index 1).

Method **`browserListView_Click`** (lines 36–84) responds when the user clicks control **`browserListView`**. Line 40 checks on whether anything is selected. If a selection has been made, line 43 determines whether the user chose the first item in **`browserListView`**. The first item in **`browserListView`** is always **Go up one level**; if it is selected, the program attempts to go up a level. Lines 46–47 create a **`DirectoryInfo`** object for the current directory. Line 50 tests property **`Parent`** to ensure that the user is not at the root of the directory tree. Property **`Parent`** indicates the parent directory as a **`DirectoryInfo`** object; if it does not exist, **`Parent`** returns the value **`null`**. If a parent directory exists, then lines 51–52 pass the full name of the parent directory to method **`LoadFilesInDirectory`**.

If the user did not select the first item in **`browserListView`**, lines 56–77 allow the user to continue navigating through the directory structure. Lines 59–60 create **`String chosen`**, which receives the text of the selected item (the first item in collection **`SelectedItems`**). Lines 63–64 test whether the user has selected a valid directory (rather than a file). The program combines variables **`currentDirectory`** and **`chosen`** (the new directory), separated by a slash (\), and passes this value to class **`Directory`**'s method ***`Exists`***. Method **`Exists`** returns **`True`** if its **`String`** parameter is a directory. If this occurs, the program passes the **`String`** to method **`LoadFilesInDirectory`**. The **`C:\`** directory already includes a slash, so a slash is not needed when combining **`currentDirectory`** and **`chosen`** (line 71). However, other directories must include the slash (lines 73–74). Finally, **`displayLabel`** is updated with the new directory (line 80).

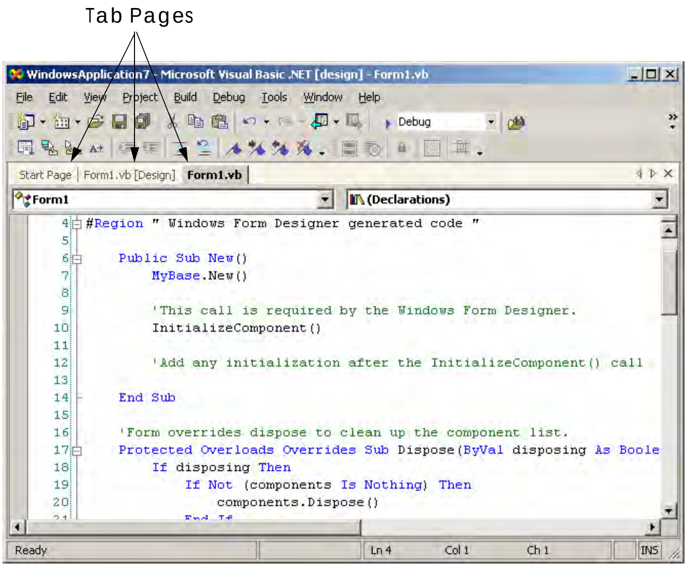
This program loads quickly, because it indexes only the files in the current directory. This means that, rather than having a large delay in the beginning, a small delay occurs whenever a new directory is loaded. In addition, changes in the directory structure can be shown by reloading a directory. The previous program (Fig. 13.21) needs to be restarted to reflect any changes in the directory structure. This type of trade-off is typical in the software world. When designing applications that run for long periods of time, developers might choose a large initial delay to improve performance throughout the rest of the program. However, when creating applications that run for only short periods of time, developers often prefer fast initial loading times and a small delay after each action.

13.8 Tab Control

The ***`TabControl`*** control creates tabbed windows, such as those we have seen in the Visual Studio .NET IDE (Fig. 13.25). This allows the programmer to design user interfaces that fit a large number of controls or a large amount of data without using up valuable screen “real estate.”

`TabControl`s contain ***`TabPage`*** objects, which are similar to **`Panel`**s and **`GroupBox`**es in that **`TabPage`**s also can contain controls. The programmer first adds controls to the **`TabPage`** objects, then adds the **`TabPage`**s to the **`TabControl`**. Only one **`TabPage`** is displayed at a time. Figure 13.26 depicts a sample **`TabControl`**.

Programmers can add **`TabControl`**s visually by dragging and dropping them onto a form in design mode. To add **`TabPage`**s in the Visual Studio .NET designer, right-click the **`TabControl`**, and select **Add Tab** (Fig. 13.27). Alternatively, click the **`TabPages`** collection in the **Properties** window, and add tabs in the dialog that appears. To change a tab label, set the **`Text`** property of the **`TabPage`**.


Tab Pages
WindowsApplication7 - Microsoft Visual Basic .NET [design] - Form1.vb
File Edit View Project Build Debug Tools Window Help
Debug
Start Page
Form1.vb [Design]
Form1.vb
Form1
(Declarations)
Ready
Ln 4
Col 1
Ch 1
INS

```
#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boole
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
```

Fig. 13.25 Tabbed pages in Visual Studio .NET.

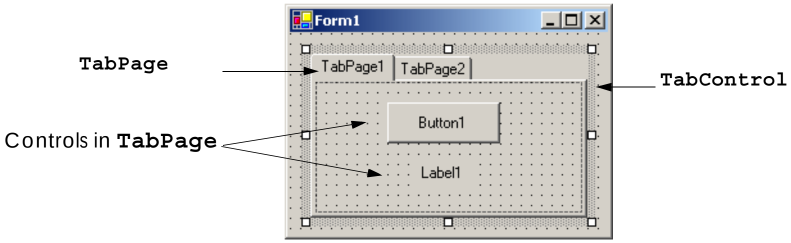

TabPage
Controls in TabPage
TabControl
Form1
TabPage1
TabPage2
Button1
Label1

Fig. 13.26 `TabControl` with `TabPage`s example.

Note that clicking the tabs selects the **`TabControl`**—to select the **`TabPage`**, click the control area underneath the tabs. The developer can add controls to the **`TabPage`** by dragging and dropping items from the **ToolBox**. To view different **`TabPage`**s, click the appropriate tab (in either design or run mode). Common properties and events of **`TabControl`**s are described in Fig. 13.28.

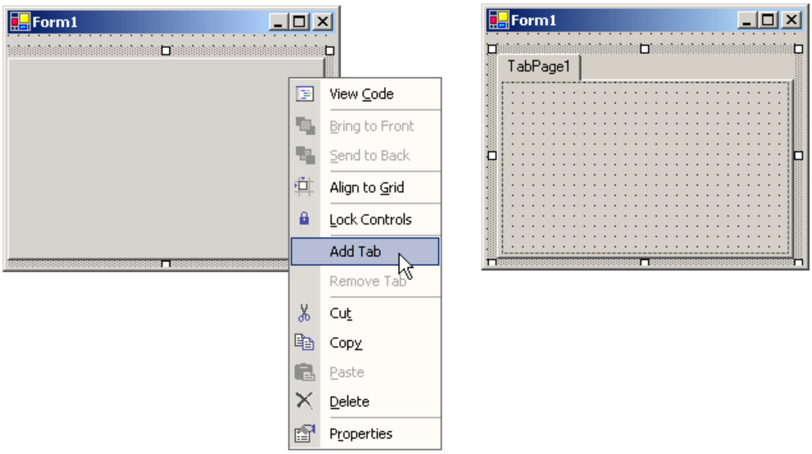

Form1
View Code
Bring to Front
Send to Back
Align to Grid
Lock Controls
Add Tab
Remove Tab
Cut
Copy
Paste
Delete
Properties
Form1
TabPage1

Fig. 13.27 `TabPage`s added to a `TabControl`.

`TabControl` properties and events	**Description / Delegate and Event Arguments**
Common Properties	
`ImageList`	Specifies images to be displayed on a tab.
`ItemSize`	Specifies tab size.
`MultiLine`	Indicates whether multiple rows of tabs can be displayed.

Fig. 13.28 `TabControl` properties and events. (Part 1 of 2.)

`TabControl` properties and events	**Description / Delegate and Event Arguments**
`SelectedIndex`	Indicates index of **`TabPage`** that is currently selected.
`SelectedTab`	Indicates the **`TabPage`** that is currently selected.
`TabCount`	Returns the number of tabs.
`TabPages`	Gets the collection of **`TabPage`**s within our **`TabControl`**.
Common Event	*(Delegate **`EventHandler`**, event arguments **`EventArgs`**)*
`SelectedIndexChanged`	Generated when **`SelectedIndex`** changes (i.e., another **`TabPage`** is selected).

Fig. 13.28 **`TabControl`** properties and events. (Part 2 of 2.)

Each **`TabPage`** generates its own **`Click`** event when its tab is clicked. Remember, events for controls can be handled by any event handler that is registered with the control's event delegate. This also applies to controls contained in a **`TabPage`**. For convenience, Visual Studio .NET generates the empty event handlers for these controls in the class in which we are currently working.

Class **`UsingTabs`** (Fig. 13.29) uses a **`TabControl`** to display various options relating to the text on a label (**Color**, **Size** and **Message**). The last **`TabPage`** displays an **About** message, which describes the use of **`TabControl`**s.

The **`TabControl optionsTabControl`** (lines 18–19) and **`TabPage`**s **`colorTabPage`** (line 22), **`sizeTabPage`** (line 30), **`messageTabPage`** (line 39) and **`aboutTabPage`** (line 46) are created in the designer (as described previously). **`TabPage colorTabPage`** contains three radio buttons—for colors black (**`blackRadioButton`**, lines 26–27), red (**`redRadioButton`**, line 25) and green (**`greenRadioButton`**, lines 23–24). The **`CheckChanged`** event handler for each button updates the color of the text in **`displayLabel`** (lines 59, 66 and 73). **`TabPage sizeTabPage`** has three radio buttons, corresponding to font sizes 12 (**`size12RadioButton`**, lines 35–36), 16 (**`size16RadioButton`**, lines 33–34) and 20 (**`size20RadioButton`**, lines 31–32), which change the font size of **`displayLabel`**—lines 80–81, 88–89 and 96–97, respectively. **`TabPage messageTabPage`** contains two radio buttons—for the messages **Hello!** (**`helloRadioButton`**, lines 42–43) and **Goodbye!** (**`goodbyeRadioButton`**, lines 40–41). The two radio buttons determine the text on **`displayLabel`** (lines 104 and 111, respectively).

```
1  // Fig. 13.29: UsingTabs.cs
2  // Using TabControl to display various font settings.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
```

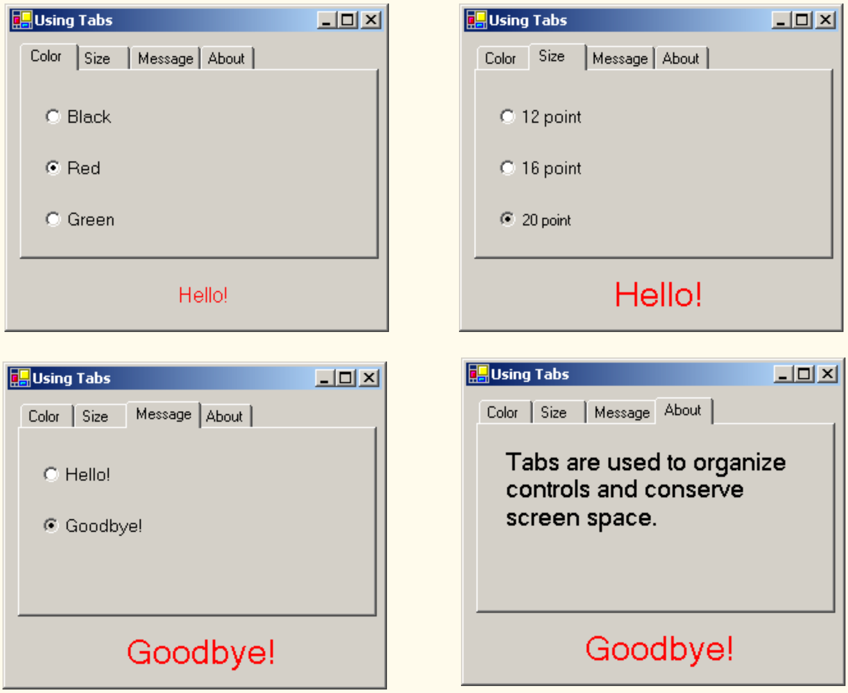
Fig. 13.29 **`TabControl`** used to display various font settings. (Part 1 of 4.)

```
using System.Data;

public class UsingTabs : System.Windows.Forms.Form
{
   // output label reflects text changes
   private System.Windows.Forms.Label displayLabel;

   // table control containing table pages colorTabPage,
   // sizeTabPage, messageTabPage and aboutTabPage
   private System.Windows.Forms.TabControl
      optionsTabControl;

   // table page containing color options
   private System.Windows.Forms.TabPage colorTabPage;
   private System.Windows.Forms.RadioButton
      greenRadioButton;
   private System.Windows.Forms.RadioButton redRadioButton;
   private System.Windows.Forms.RadioButton
      blackRadioButton;

   // table page containing font size options
   private System.Windows.Forms.TabPage sizeTabPage;
   private System.Windows.Forms.RadioButton
      size20RadioButton;
   private System.Windows.Forms.RadioButton
      size16RadioButton;
   private System.Windows.Forms.RadioButton
      size12RadioButton;

   // table page containing text display options
   private System.Windows.Forms.TabPage messageTabPage;
   private System.Windows.Forms.RadioButton
      goodByeRadioButton;
   private System.Windows.Forms.RadioButton
      helloRadioButton;

   // table page containing about message
   private System.Windows.Forms.TabPage aboutTabPage;
   private System.Windows.Forms.Label messageLabel;

   [STAThread]
   static void Main()
   {
      Application.Run( new UsingTabs() );
   }

   // event handler for black color radio button
   private void blackRadioButton_CheckedChanged(
      object sender, System.EventArgs e )
   {
      displayLabel.ForeColor = Color.Black;
   }

```

Fig. 13.29 `TabControl` used to display various font settings. (Part 2 of 4.)

```
      // event handler for red color radio button
      private void redRadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.ForeColor = Color.Red;
      }

      // event handler for green color radio button
      private void greenRadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.ForeColor = Color.Green;
      }

      // event handler for size 12 radio button
      private void size12RadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.Font =
            new Font( displayLabel.Font.Name, 12 );
      }

      // event handler for size 16 radio button
      private void size16RadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.Font =
            new Font( displayLabel.Font.Name, 16 );
      }

      // event handler for size 20 radio button
      private void size20RadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.Font =
            new Font( displayLabel.Font.Name, 20 );
      }

      // event handler for message "Hello!" radio button
      private void helloRadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.Text = "Hello!";
      }

      // event handler for message "Goodbye!" radio button
      private void goodByeRadioButton_CheckedChanged(
         object sender, System.EventArgs e )
      {
         displayLabel.Text = "Goodbye!";
      }

   } // end class UsingTabs
```

Fig. 13.29 `TabControl` used to display various font settings. (Part 3 of 4.)



Tabs are used to organize controls and conserve screen space.

Fig. 13.29 `TabControl` used to display various font settings. (Part 4 of 4.)

Software Engineering Observation 13.2

A **`TabPage`** *can act as a container for a single logical group of radio buttons and enforces their mutual exclusivity. To place multiple radio-button groups inside a single* **`TabPage`**, *programmers should group radio buttons within* **`Panel`***s or* **`GroupBox`***es contained within the* **`TabPage`**.

The last **`TabPage`** (**`aboutTabPage`**, line 46) contains a **`Label`** (**`messageLabel`**, line 47) that describes the purpose of **`TabControl`**s.

13.9 Multiple-Document-Interface (MDI) Windows

In previous chapters, we have built only *single-document-interface (SDI)* applications. Such programs (including Notepad or Paint) support only one open window or document at a time. SDI applications usually have contracted abilities—Paint and Notepad, for example, have limited image- and text-editing features. To edit multiple documents, the user must create additional instances of the SDI application.

Multiple document interface (MDI) programs (such as PaintShop Pro and Adobe Photoshop) enable users to edit multiple documents at once. MDI programs also tend to be more complex—PaintShop Pro and Photoshop have a greater number of image-editing features than does Paint. Until now, we had not mentioned that the applications we created were SDI applications. We define this here to emphasize the distinction between the two types of programs.

The application window of an MDI program is called the *parent window*, and each window inside the application is referred to as a *child window*. Although an MDI application can have many child windows, each has only one parent window. Furthermore, a maximum of one child window can be active at once. Child windows cannot be parents themselves and cannot be moved outside their parent. Otherwise, a child window behaves like any other window (with regard to closing, minimizing, resizing etc.). A child window's functionality can be different from the functionality of other child windows of the parent. For example, one child window might edit images, another might edit text and a third might display network traffic graphically, but all could belong to the same MDI parent. Figure 13.30 depicts a sample MDI application.

To create an MDI form, create a new **`Form`** and set its ***`IsMDIContainer`*** property to **`True`**. The form changes appearance, as in Fig. 13.31.

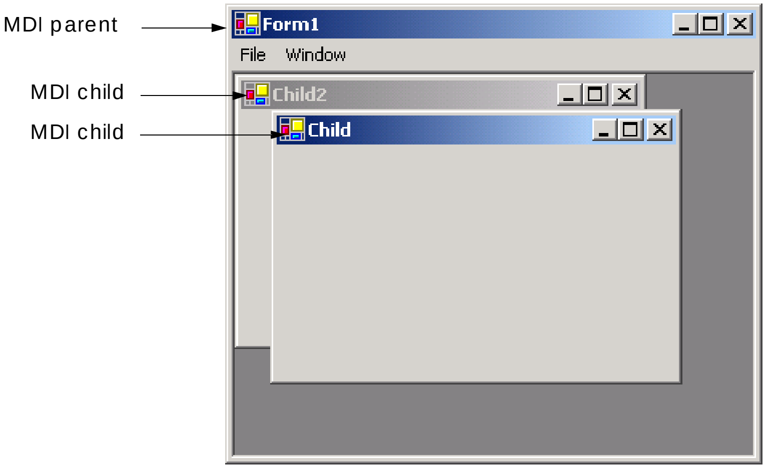

MDI parent
MDI child
MDI child
Form1
File
Window
Child2
Child

Fig. 13.30 MDI parent window and MDI child windows.

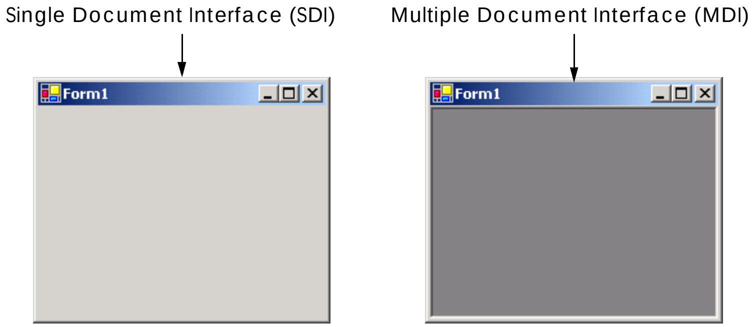

Single Document Interface (SDI)
Multiple Document Interface (MDI)
Form1
Form1

Fig. 13.31 SDI and MDI forms.

Next, create a child form class to be added to the form. To do this, right-click the project in the **Solution Explorer**, select **Add Windows Form...** and name the file. To add the child form to the parent, we must create a new child form object; set its **`MdiParent`** property to the parent form, and call method **`Show`**. The code to create a child usually lies inside an event handler, which creates a new window in response to a user action. Menu selections (such as **File** followed by a submenu option of **New** followed by a submenu option of **Window**) are common methods of creating new child windows.

`Form` property ***`MdiChildren`*** is an array of child **`Form`** references. This is useful if the parent window wants to check the status of all its children (such as to ensure that all are saved before the parent closes). Property ***`ActiveMdiChild`*** returns a reference to the active child window; it returns **`null`** if there are no active child windows. Other features of MDI windows are described in Fig. 13.32.

Child windows can be minimized, maximized and closed independently of each other and of the parent window. Figure 13.33 shows two images, one containing two minimized child windows and a second containing a maximized child window. When the parent is minimized or closed, the child windows are minimized or closed as well. Notice that the title bar in the second image of Fig. 13.33 is **Parent Window - [Child]**. When a child window is maximized, its title bar is inserted into the parent window's title bar. When a child window is minimized or maximized, its title bar displays a restore icon, which returns the child window to its previous size (its size before it was minimized or maximized).

MDI `Form` events and properties	**Description / Delegate and Event Arguments**
Common MDI Child Properties	
`IsMdiChild`	Indicates whether the **`Form`** is an MDI child. If **`True`**, **`Form`** is an MDI child (read-only property).
`MdiParent`	Specifies the MDI parent **`Form`** of the child.
Common MDI Parent Properties	
`ActiveMdiChild`	Returns the **`Form`** that is the currently active MDI child (returns **`null`** if no children are active).
`IsMdiContainer`	Indicates whether a **`Form`** can be an MDI. If **`True`**, the **`Form`** can be an MDI parent. Default is **`False`**.
`MdiChildren`	Returns the MDI children as an array of **`Form`**s.
Common Method	
`LayoutMdi`	Determines the display of child forms on an MDI parent. Takes as a parameter an **`MdiLayout`** enumeration with possible values **`ArrangeIcons`**, **`Cascade`**, **`TileHorizontal`** and **`TileVertical`**. Figure 13.35 depicts the effects of these values.
Common Event	*(Delegate* ***`EventHandler`****, event arguments* ***`EventArgs`****)*
`MdiChildActivate`	Generated when an MDI child is closed or activated.

Fig. 13.32 MDI parent and MDI child events and properties.

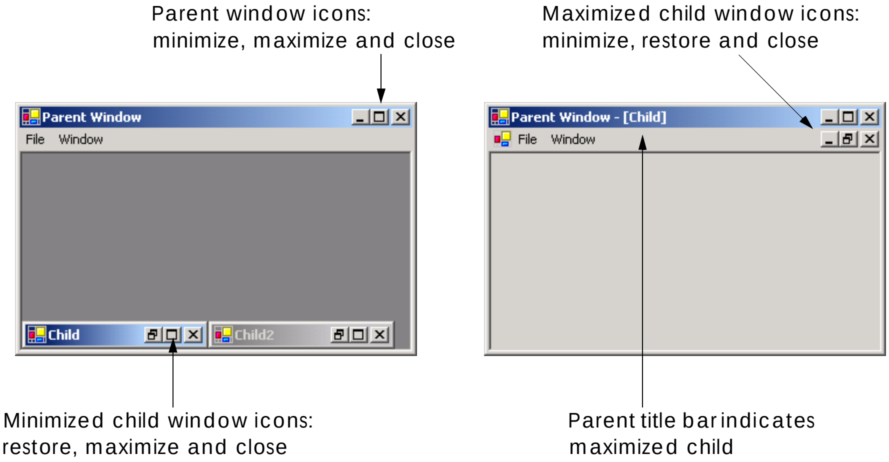

Parent window icons:
minimize, maximize and close
Maximized child window icons:
minimize, restore and close
Parent Window
File Window
Child
Child2
Parent Window - [Child]
File Window
Minimized child window icons:
restore, maximize and close
Parent title bar indicates
maximized child

Fig. 13.33 Minimized and maximized child windows.

The parent and child forms can have different menus, which are merged whenever a child window is selected. To specify how the menus merge, programmers can set the ***`MergeOrder`*** and the ***`MergeType`*** properties for each **`MenuItem`** (see Fig. 13.3). **`MergeOrder`** determines the order in which **`MenuItem`**s appear when two menus are merged. **`MenuItem`**s with a lower **`MergeOrder`** value will appear first. For example, if **`Menu1`** has items **File**, **Edit** and **Window** (and their orders are 0, 10 and 20) and **`Menu2`** has items **Format** and **View** (and their orders are 7 and 15), then the merged menu contains menu items **File**, **Format**, **Edit**, **View** and **Window**, in that order.

Each **`MenuItem`** instance has its own **`MergeOrder`** property. It is likely that, at some point in an application, two **`MenuItem`**s with the same **`MergeOrder`** value will merge. Property **`MergeType`** resolves this conflict by following the order in which the two menus are displayed.

The **`MergeType`** property takes a ***`MenuMerge`*** enumeration value and determines which menu items will be displayed when two menus are merged. A menu item with value ***`Add`*** is added to its parent's menu as a new menu on the menu bar (the parent's menu items come first). If a child form's menu item has value ***`Replace`***, it attempts to take the place of its parent form's corresponding menu item during merging. A menu with value ***`MergeItems`*** combines its items with that of its parent's corresponding menu (if parent and child menus originally occupy the same space, their submenus will be brought together as one large menu). A child's menu item with value ***`Remove`*** disappears when the menu is merged with that of its parent.

Value **`MergeItems`** acts passively—if the parent's menu has a **`MergeType`** that is different from the child menu's **`MergeType`**, the child's menu setting determines the outcome of the merge. When the child window is closed, the parent's original menu is restored.

Good Programming Practice 13.1

When creating MDI applications, include a menu item with its ***`MdiList`*** *property set to* ***`True`****. This helps the user select a child window quickly, rather than having to search for it in the parent window.*

Software Engineering Observation 13.3

Set the parent's menu items' **`MergeType`** *property to value* **`MergeItems`**. *This allows the child window to add most menu items according to its own settings. Parent menu items that must remain should have value* **`Add`**, *and those that must be removed should have value* **`Remove`**.

C# provides a property that facilitates the tracking of which child windows are opened in an MDI container. Property ***`MdiList`*** (a boolean) of class **`MenuItem`** determines whether a **`MenuItem`** displays a list of open child windows. The list appears at the bottom of the menu following a separator bar (first screen in Figure 13.34). When a new child window is opened, an entry is added to the list. If nine or more child windows are open, the list includes the option **More Windows...**, which allows the user to select a window from a list, using a scrollbar. Multiple **`MenuItem`**s can have their **`MdiList`** property set; each displays a list of open child windows.

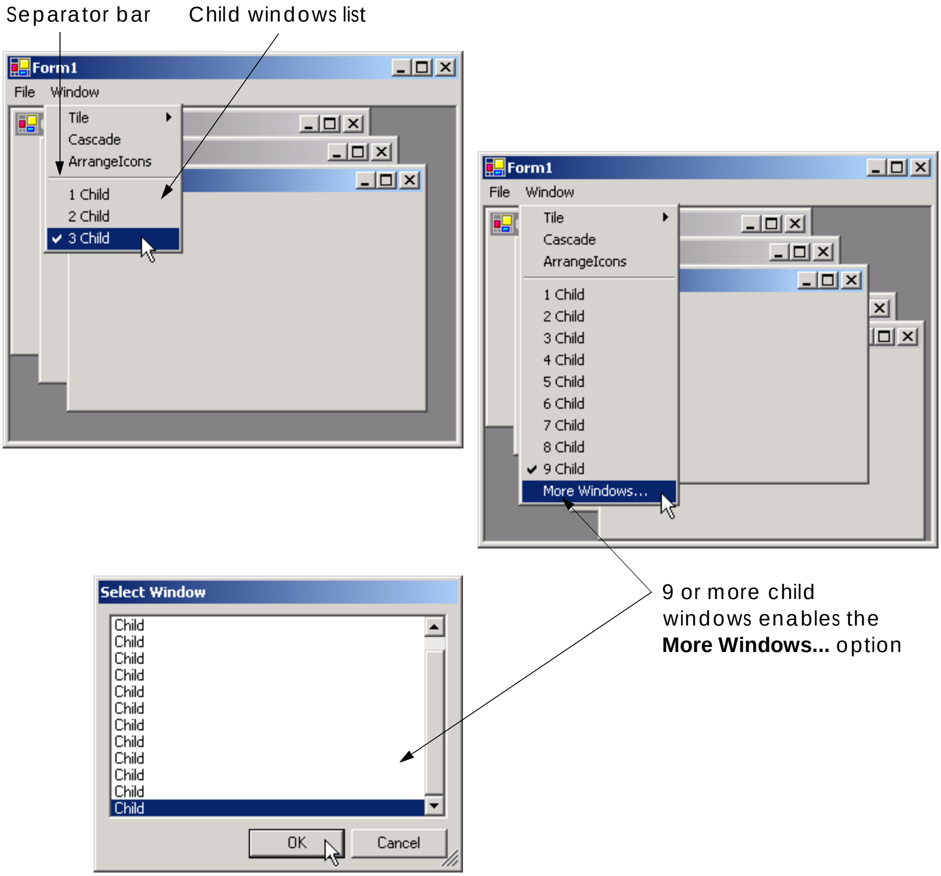
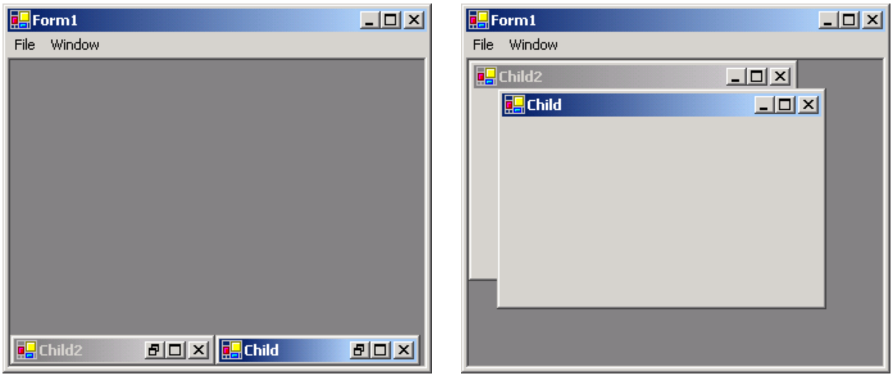

Separator bar
Child windows list
Form1
File
Window
Tile
Cascade
ArrangeIcons
1 Child
2 Child
3 Child
Form1
File
Window
Tile
Cascade
ArrangeIcons
1 Child
2 Child
3 Child
4 Child
5 Child
6 Child
7 Child
8 Child
9 Child
More Windows...
9 or more child windows enables the **More Windows...** option
Select Window
Child
Child
Child
Child
Child
Child
Child
Child
Child
Child
Child
Child
OK
Cancel

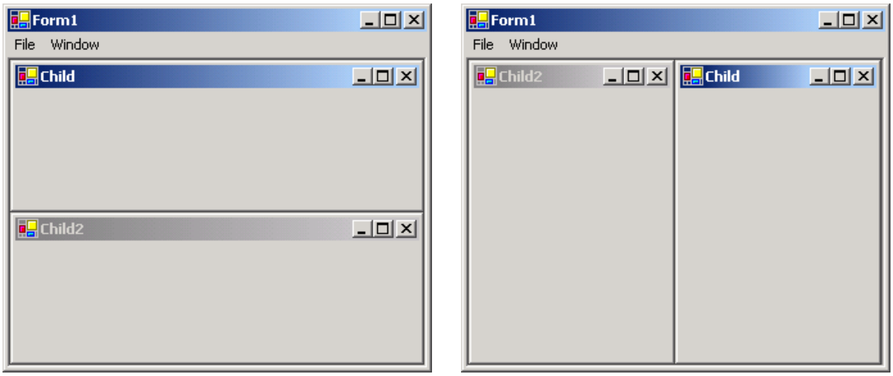
Fig. 13.34 **`MenuItem`** property **`MdiList`** example.

MDI containers allow developers to organize the placement of child windows. The child windows in an MDI application can be arranged by calling method ***`LayoutMdi`*** of the parent form. Method **`LayoutMdi`** takes a ***`LayoutMdi`*** *enumeration*, which can have values ***`ArrangeIcons`***, ***`Cascade`***, ***`TileHorizontal`*** and ***`TileVertical`***. *Tiled windows* completely fill the parent and do not overlap; such windows can be arranged horizontally (value **`TileHorizontal`**) or vertically (value **`TileVertical`**). *Cascaded windows* (value **`Cascade`**) overlap—each is the same size and displays a visible title bar, if possible. Value **`ArrangeIcons`** arranges the icons for any minimized child windows. If minimized windows are scattered around the parent window, value **`ArrangeIcons`** orders them neatly at the bottom-left corner of the parent window. Figure 13.35 illustrates the values of the **`LayoutMdi`** enumeration.

Class **`UsingMDI`** (Fig. 13.36) demonstrates the use of MDI windows. Class **`UsingMdi`** uses three instances of class **`Child`** (Fig. 13.37), each of which contains a **`PictureBox`** and an image of a book cover. The parent MDI form contains a menu that enables users to create and arrange child forms.



`ArrangeIcons` **`Cascade`**



`TileHorizontal` **`TileVertical`**

Fig. 13.35 `LayoutMdi` enumeration values.

The MDI parent form (Fig. 13.36) contains two top-level menus. The first of these menus, **File** (`fileMenuItem`, line 13), contains both an **Exit** item (`exitMenuItem`, line 18) and a **New** submenu (`newMenuItem`, line 14) consisting of items for each child window. The second menu, **Format** (`formatMenuItem`, line 19), provides options for laying out the MDI children, plus a list of the active MDI children.

In the **Properties** window, we set the `Form`'s `IsMdiContainer` property to `True`, making the `Form` an MDI parent. In addition, we set `formatMenuItem` property `MdiList` to `True`. This enables `formatMenuItem` to list the active child MDI windows.

```
 1  // Fig. 13.36: UsingMDI.cs
 2  // Demonstrating use of MDI parent and child windows.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.Data;
 9
10  public class UsingMDI : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.MainMenu mainMenu1;
13     private System.Windows.Forms.MenuItem fileMenuItem;
14     private System.Windows.Forms.MenuItem newMenuItem;
15     private System.Windows.Forms.MenuItem child1MenuItem;
16     private System.Windows.Forms.MenuItem child2MenuItem;
17     private System.Windows.Forms.MenuItem child3MenuItem;
18     private System.Windows.Forms.MenuItem exitMenuItem;
19     private System.Windows.Forms.MenuItem formatMenuItem;
20     private System.Windows.Forms.MenuItem cascadeMenuItem;
21     private System.Windows.Forms.MenuItem
22        tileHorizontalMenuItem;
23     private System.Windows.Forms.MenuItem
24        tileVerticalMenuItem;
25
26     [STAThread]
27     static void Main()
28     {
29        Application.Run( new UsingMDI() );
30     }
31
32     // create Child 1 when menu clicked
33     private void child1MenuItem_Click(
34        object sender, System.EventArgs e )
35     {
36        // create new child
37        Child formChild = new Child( "Child 1",
38           "\\images\\csharphtp1.jpg" );
39        formChild.MdiParent = this;    // set parent
40        formChild.Show();              // display child
41     }
42
```

Fig. 13.36 MDI parent-window class. (Part 1 of 3.)

```
   // create Child 2 when menu clicked
   private void child2MenuItem_Click(
      object sender, System.EventArgs e )
   {
      // create new child
      Child formChild = new Child( "Child 2",
         "\\images\\vbnethtp2.jpg" );
      formChild.MdiParent = this;   // set parent
      formChild.Show();             // display child
   }

   // create Child 3 when menu clicked
   private void child3MenuItem_Click(
      object sender, System.EventArgs e )
   {
      // create new child
      Child formChild = new Child( "Child 3",
         "\\images\\pythonhtp1.jpg" );
      formChild.MdiParent = this;   // set parent
      formChild.Show();             // display child
   }

   // exit application
   private void exitMenuItem_Click(
      object sender, System.EventArgs e )
   {
      Application.Exit();
   }

   // set cascade layout
   private void cascadeMenuItem_Click(
      object sender, System.EventArgs e )
   {
      this.LayoutMdi( MdiLayout.Cascade );
   }

   // set TileHorizontal layout
   private void tileHorizontalMenuItem_Click(
      object sender, System.EventArgs e )
   {
      this.LayoutMdi( MdiLayout.TileHorizontal );
   }

   // set TileVertical layout
   private void tileVerticalMenuItem_Click(
      object sender, System.EventArgs e )
   {
      this.LayoutMdi( MdiLayout.TileVertical );
   }

} // end class UsingMDI
```

Fig. 13.36 MDI parent-window class. (Part 2 of 3.)

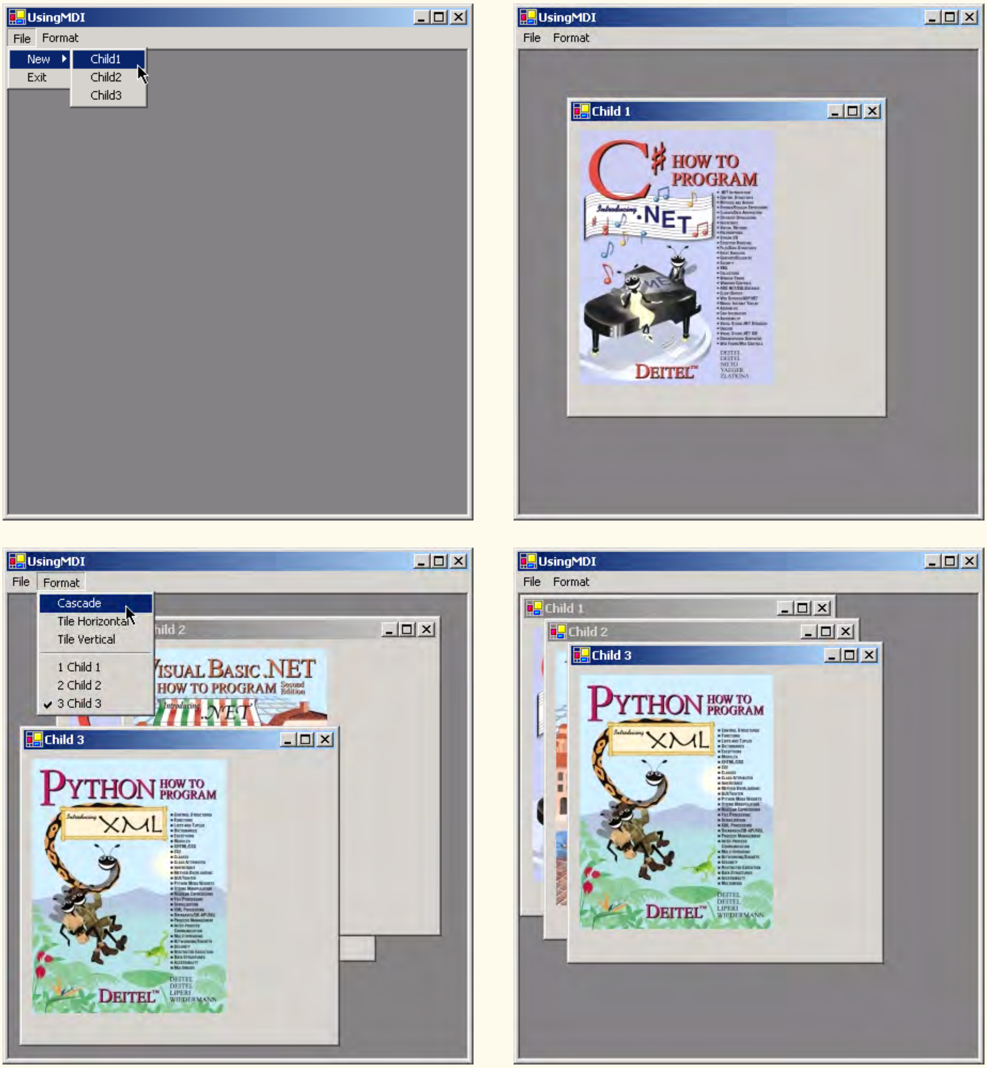

UsingMDI
File
Format
New
Exit
Child1
Child2
Child3
Child 1
Cascade
Tile Horizontal
Tile Vertical
1 Child 1
2 Child 2
✓ 3 Child 3
Child 2
Child 3

Fig. 13.36 MDI parent-window class. (Part 3 of 3.)

The **Cascade** menu item (`cascadeMenuItem`, line 20) has an event handler (`cascadeMenuItem_Click`, lines 73–77) that arranges the child windows in a cascading manner. The event handler calls method `LayoutMdi` with the argument `Cascade` from the `MdiLayout` enumeration (line 76).

The **Tile Horizontal** menu item (`tileHorizontalMenuItem`, lines 21–22) has an event handler (`mnuitmTileHorizontal_Click`, lines 80–84) that arranges the child windows in a horizontal manner. The event handler calls method `LayoutMdi` with the argument `TileHorizontal` from the `MdiLayout` enumeration (line 83).

Finally, the **Tile Vertical** menu item (`mnuitmTileVertical`, lines 23–24) has an event handler (`mnuitmTileVertical_Click`, lines 87–91) that arranges the child windows in a vertical manner. The event handler calls method **`LayoutMdi`** with the argument **`TileVertical`** from the **`MdiLayout`** enumeration (line 90).

To define the child class for the MDI application, right-click the project in the **Solution Explorer** and select first **Add** and then **Add Windows Form...**. Name the new class **`Child`** (Fig. 13.37).

Next, we add a **`PictureBox`** (**`picDisplay`**, line 11) to form **`Child`**. The constructor invokes method **`InitializeComponent`** (line 17) and initializes the form's title (line 19) and the image to display in the **`PictureBox`** (lines 22–23).

The parent MDI form (Fig. 13.36) creates new instances of class **`Child`** each time the user selects a new child window from the **File** menu. The event handlers in lines 33–63 create new child forms that contain images of Deitel and Associates, Inc. book covers. Each event handler creates a new instance of the child form, sets its **`MdiParent`** property to the parent form and calls method **`Show`** to display the child.

13.10 Visual Inheritance

In Chapter 9, Object-Oriented Programming: Inheritance, we discuss how to create classes by inheriting from other classes. In C#, we also can use inheritance to create **`Form`**s that display a GUI, because **`Form`**s are classes that derive from class **`System.Windows.Forms.Form`**. Visual inheritance allows us to create a new **`Form`** by inheriting from another **`Form`**. The derived **`Form`** class contains the functionality of its **`Form`** base class, including any base-class properties, methods, variables and controls. The derived class also inherits all visual aspects—such as sizing, component layout, spacing between GUI components, colors and fonts—from its base class.

Visual inheritance enables developers to achieve visual consistency across applications by reusing code. For example, a company could define a base form that contains a product's logo, a static background color, a predefined menu bar and other elements. Programmers then could use the base form throughout an application for purposes of uniformity and product branding.

```
 1  // Fig. 13.37: Child.cs
 2  // Child window of MDI parent.
 3  using System;
 4  using System.Drawing;
 5  using System.Collections;
 6  using System.ComponentModel;
 7  using System.Windows.Forms;
 8  using System.IO;
 9
10  public class Child : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.PictureBox pictureBox;
13
14     public Child( string title, string fileName )
15     {
16        // Required for Windows Form Designer support
17        InitializeComponent();
```

Fig. 13.37 **`Child`** class for MDI demonstration. (Part 1 of 2.)

```

      Text = title; // set title text

      // set image to display in pictureBox
      pictureBox.Image = Image.FromFile(
         Directory.GetCurrentDirectory() + fileName );
   }
}
```

Fig. 13.37 `Child` class for MDI demonstration. (Part 2 of 2.)

Class `VisualInheritance` (Fig. 13.38) is a form that we use as a base class for demonstrating visual inheritance. The GUI contains two labels (one with text **Bugs, Bugs**, **Bugs** and one with **Copyright 2002, by Bug2Bug.com.**) and one button (displaying the text **Learn More**). When a user presses the **Learn More** button, method `learnMoreButton_Click` (lines 22–29) is invoked. This method displays a message box that provides some informative text.

```
1  // Fig. 13.38: VisualInheritance.cs
2  // Base Form for use with visual inheritance
3  using System;
4  using System.Drawing;
5  using System.Collections;
6  using System.ComponentModel;
7  using System.Windows.Forms;
8  using System.Data;
9
10 public class VisualInheritance : System.Windows.Forms.Form
11 {
12    private System.Windows.Forms.Label bugsLabel;
13    private System.Windows.Forms.Button learnMoreButton;
14    private System.Windows.Forms.Label label1;
15
16    [STAThread]
17    static void Main()
18    {
19       Application.Run( new VisualInheritance() );
20    }
21
22    private void learnMoreButton_Click( object sender,
23       System.EventArgs e )
24    {
25       MessageBox.Show(
26          "Bugs, Bugs, Bugs is a product of Bug2Bug.com",
27          "Learn More", MessageBoxButtons.OK,
28          MessageBoxIcon.Information );
29    }
30 }
```

Fig. 13.38 Class `FrmInheritance`, which inherits from class `Form`, contains a button (**Learn More**). (Part 1 of 2.)

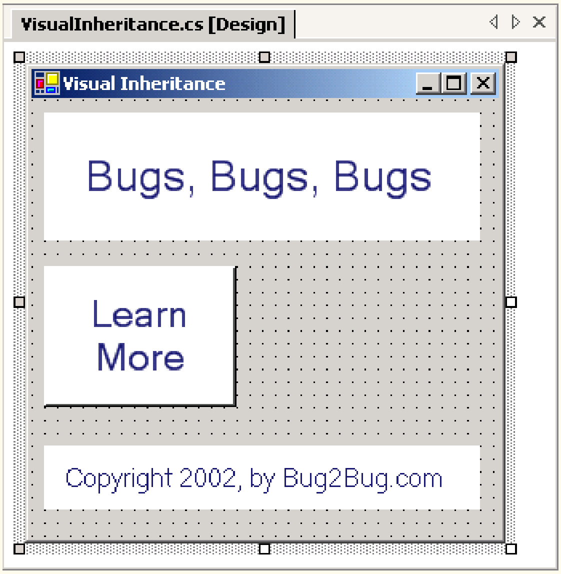


Fig. 13.38 Class `FrmInheritance`, which inherits from class `Form`, contains a button (**Learn More**). (Part 2 of 2.)

Before deriving a form from class `VisualInheritance`, we must package class `VisualInheritance` in a `.dll`. Right click on the `VisualInheritance` project in the **Solution Explorer**, and select **Properties**. In **Common Properties > General**, change the **Output Type** to **Class Library**. Then, build the project to produce a `.dll` that contains the `VisualInheritance` class.

To create the derived form through visual inheritance, create an empty project. From the **Project** menu, select **Add Inherited Form...**. This brings up the **Add New Item** window. Select **Inherited Form** from the templates window. Clicking **OK** displays the **Inheritance Picker**. The **Inheritance Picker** tool enables programmers to quickly create a form that inherits from a specified form. Click **Browse**, and select the `.dll` file for class `VisualInheritance`. The `.dll` file normally is located within the `bin\Debug` directory of the `VisualInheritance` project directory. Click **OK**. The Form Designer should now display the inherited form (Fig. 13.39).

Class `VisualInheritanceTest` (Fig. 13.40) derives from class `VisualInheritance`. The GUI contains those components derived from class `VisualInheritance`, plus a button with text **Learn The Program** that we added in class `VisualInheritanceTest`. When a user presses this button, method `learnProgramButton_Click` (lines 15–22) is invoked. This method displays a simple message box.

Figure 13.40 demonstrates that the components, their layouts and the functionality of the base class `VisualInheritance` (Fig. 13.38) are inherited by `VisualInheritanceTest`. If a user clicks button **Learn More**, the base-class event handler `learnMoreButton_Click` displays a `MessageBox`.

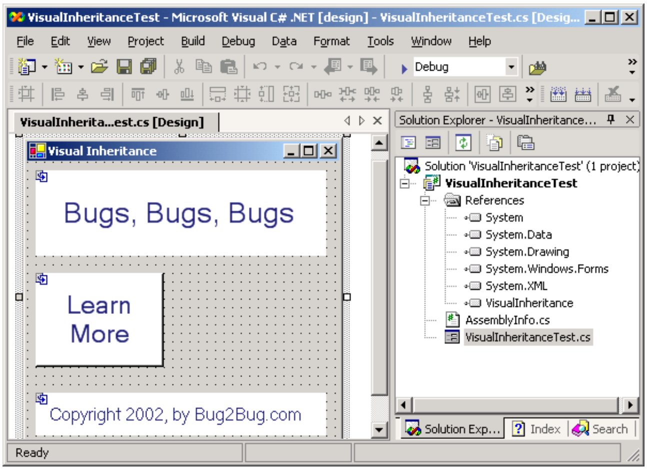
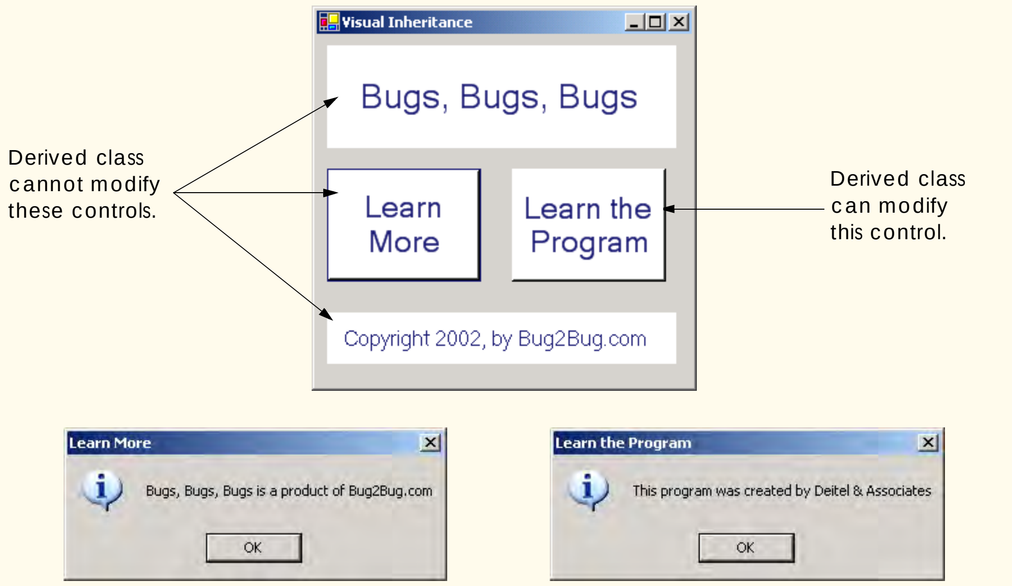

VisualInheritanceTest - Microsoft Visual C# .NET [design] - VisualInheritanceTest.cs [Desig...
File Edit View Project Build Debug Data Format Tools Window Help
Debug
VisualInherita...est.cs [Design]
Visual Inheritance
Bugs, Bugs, Bugs
Learn More
Copyright 2002, by Bug2Bug.com
Solution Explorer - VisualInheritance...
Solution 'VisualInheritanceTest' (1 project)
VisualInheritanceTest
References
System
System.Data
System.Drawing
System.Windows.Forms
System.XML
VisualInheritance
AssemblyInfo.cs
VisualInheritanceTest.cs
Solution Exp...
Index
Search
Ready

Fig. 13.39 Visual Inheritance through the Form Designer.

```
// Fig. 13.40: VisualInheritanceTest.cs
// Derived Form using visual inheritance.
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;

public class VisualInheritanceTest :
   VisualInheritance.VisualInheritance
{
   private System.Windows.Forms.Button learnProgramButton;

   // invoke when user clicks Learn the Program Button
   private void learnProgramButton_Click( object sender,
      System.EventArgs e )
   {
      MessageBox.Show(
         "This program was created by Deitel & Associates",
         "Learn the Program", MessageBoxButtons.OK,
         MessageBoxIcon.Information );
   }

   public static void Main( string[] args )
   {
      Application.Run( new VisualInheritanceTest() );
   }
}
```

Fig. 13.40 Class `FrmVisualTest`, which inherits from class `VisualForm.FrmInheritance`, contains an additional button. (Part 1 of 2.)



Derived class cannot modify these controls.

Derived class can modify this control.

Bugs, Bugs, Bugs is a product of Bug2Bug.com

This program was created by Deitel & Associates

Fig. 13.40 Class **`FrmVisualTest`**, which inherits from class **`VisualForm.FrmInheritance`**, contains an additional button. (Part 2 of 2.)

13.11 User-Defined Controls

The .NET Framework allows programmers to create *custom controls* that inherit from a variety of classes. These custom controls appear in the user's **Toolbox** and can be added to **`Form`**s, **`Panel`**s or **`GroupBox`**es in the same way that we add **`Button`**s, **`Label`**s, and other predefined controls. The simplest way to create a custom control is to derive a class from an existing Windows Forms control, such as a **`Label`**. This is useful if the programmer wants to include functionality of an existing control, rather than having to reimplement the existing control in addition to including the desired new functionality. For example, we can create a new type of label that behaves like a normal **`Label`** but has a different appearance. We accomplish this by inheriting from class **`Label`** and overriding method **`OnPaint`**.

Look-and-Feel Observation 13.8

To change the appearance of any control, override method ***`OnPaint`****.*

All controls contain method **`OnPaint`**, which the system calls when a component must be redrawn (such as when the component is resized). Method **`OnPaint`** is passed a ***`PaintEventArgs`*** object, which contains graphics information—property **`Graphics`** is the graphics object used to draw, and property ***`ClipRectangle`*** defines the rectangular boundary of the control. Whenever the system generates the **`Paint`** event, our control's base class catches the event. Through polymorphism, our control's **`OnPaint`** method is called. Our base class's **`OnPaint`** implementation is not called, so we must call it explicitly from our **`OnPaint`** implementation before we execute our custom-paint code. Alternatively, if we do not wish to let our base class paint itself, we should not call our base class's **`OnPaint`** method implementation.

To create a new control composed of existing controls, use class ***`UserControl`***. Controls added to a custom control are called *constituent controls*. For example, a programmer could create a **`UserControl`** composed of a button, a label and a text box, each associated with some functionality (such as that the button sets the label's text to that contained in the text box). The **`UserControl`** acts as a container for the controls added to it. The **`UserControl`** contains constituent controls, so it does not determine how these constituent controls are displayed. Method **`OnPaint`** cannot be overridden in these custom controls—their appearance can be modified only by handling each constituent control's **`Paint`** event. The **`Paint`** event handler is passed a **`PaintEventArgs`** object, which can be used to draw graphics (lines, rectangles etc.) on the constituent controls.

Using another technique, a programmer can create a brand-new control by inheriting from class **`Control`**. This class does not define any specific behavior; that task is left to the programmer. Instead, class **`Control`** handles the items associated with all controls, such as events and sizing handles. Method **`OnPaint`** should contain a call to the base class's **`OnPaint`** method, which calls the **`Paint`** event handlers. The programmer must then add code for custom graphics inside the overridden **`OnPaint`** method. This technique allows for the greatest flexibility, but also requires the most planning. All three approaches are summarized in Fig. 13.41.

Custom Control Techniques and `PaintEventArgs` Properties	**Description**
Inherit from Windows Forms control	Add functionality to a preexisting control. If overriding method **`OnPaint`**, call base class **`OnPaint`**. Can only add to the original control appearance, not redesign it.
*Create a **`UserControl`***	Create a **`UserControl`** composed of multiple preexisting controls (and combine their functionality). Cannot override **`OnPaint`** methods of custom controls. Instead, add drawing code to a **`Paint`** event handler. Can only add to the original control appearance, not redesign it.
*Inherit from class **`Control`***	Define a brand-new control. Override **`OnPaint`** method, call base class method **`OnPaint`** and include methods to draw the control. Can customize control appearance and functionality.
***`PaintEventArgs`** Properties*	*Use this object inside method **`OnPaint`** or **`Paint`** to draw on the control.*
`Graphics`	Indicates the graphics object of control. Used to draw on control.
`ClipRectangle`	Specifies the rectangle indicating boundary of control.

Fig. 13.41 Custom control creation.

We create a “clock” control in Fig. 13.42. This is a `UserControl` composed of a label and a timer—whenever the timer generates an event, the label is updated to reflect the current time.

`Timer`s (namespace `System.Windows.Forms`) are invisible components that reside on a form and generate ***`Tick`*** events at a set interval. This interval is set by the `Timer`’s ***`Interval`*** property, which defines the number of milliseconds (thousandths of a second) between events. By default, timers are disabled.

We create a `Form` that displays our custom control, `ClockUserControl` (Fig. 13.42). Create a `UserControl` class for the project by selecting **Project** > **Add User Control...**. This displays a dialog from which we can select the type of control to add—user controls are already selected. We then name the file (and the class) `ClockUserControl`. This brings up our empty `ClockUserControl` as a grey rectangle.

We can treat this control like a Windows `Form`, so we can add controls (using the **ToolBox**) and set properties (using the **Properties** window). However, instead of creating an application (notice there is no `Main` method in the `Control` class), we are simply creating a new control composed of other controls. We add a `Label` (`displayLabel`, line 15) and a `Timer` (`clockTimer`, line 14) to the `UserControl`. We set the `Timer` interval to 100 milliseconds and update `displayLabel`’s text with each event (lines 18–24). Note that `clockTimer` must be enabled by setting property `Enabled` to `True` in the **Properties** window.

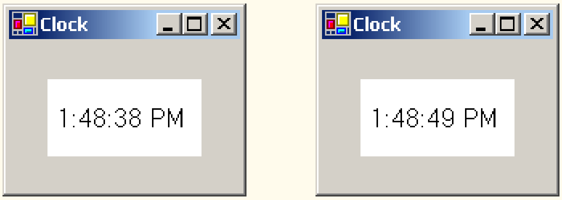
Structure `DateTime` (namespace `System`) contains member `Now`, which is the current time. Method `ToLongTimeString` converts `Now` to a `String` that contains the current hour, minute, and second (along with AM or PM). We use this to set `displayLabel`’s `Text` property on line 22.

Once created, our clock control appears as an item on the **ToolBox**. To use the control, we can simply drag it onto a Windows application in our project and run the Windows application. The `ClockUserControl` object has a white background to make it stand out in the form. Figure 13.42 shows the output of `ClockExample`, which is a simple form that contains our `ClockUserControl`.

```
1  // Fig. 13.42: ClockUserControl.cs
2  // User-defined control with a timer and a label.
3
4  using System;
5  using System.Collections;
6  using System.ComponentModel;
7  using System.Drawing;
8  using System.Data;
9  using System.Windows.Forms;
10
11 public class ClockUserControl
12    : System.Windows.Forms.UserControl
13 {
14    private System.Windows.Forms.Timer clockTimer;
15    private System.Windows.Forms.Label displayLabel;
16
```

Fig. 13.42 Programmer-defined control that displays the current time. (Part 1 of 2.)

```
      // update label at every tick
      private void clockTimer_Tick(
         object sender, System.EventArgs e )
      {
         // get current time (Now), convert to string
         displayLabel.Text = DateTime.Now.ToLongTimeString();

      } // end method clockTimer_Tick

   } // end class ClockUserControl
```



```
1:48:38 PM
```

```
1:48:49 PM
```

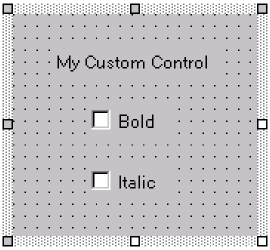
Fig. 13.42 Programmer-defined control that displays the current time. (Part 2 of 2.)

The above steps are useful when we need to define a custom control for the project on which we are working. Visual Studio .NET allows developers to share their custom controls with other developers. To create a `UserControl` that can be exported to other solutions, do the following:

1. Create a new **Windows Control Library** project.
2. Inside the project, add controls and functionality to the `UserControl` (Fig. 13.43).
3. Build the project. Visual Studio .NET creates a `.dll` file for the `UserControl` in the output directory. The file is not executable: `Control` classes do not have a `Main` method. Select **Project** > **Properties** to find the output directory and output file (Fig. 13.44).
4. Create a new Windows application.
5. Import the `UserControl`. In the new Windows application, right click the **ToolBox**, and select **Customize Toolbox...**. In the dialog that appears, select the **.NET Framework Components** tab. Browse for the `.dll` file, which is in the output directory for the Windows control library project. Click the checkbox next to the control, and click **OK** (Fig. 13.45).
6. The `UserControl` appears on the **ToolBox** and can be added to the form as if it were any other control (Fig. 13.46).

Testing and Debugging Tip 13.1

Control classes do not have a `Main` method—they cannot be run by themselves. To test their functionality, add them to a sample Windows application and run them there.

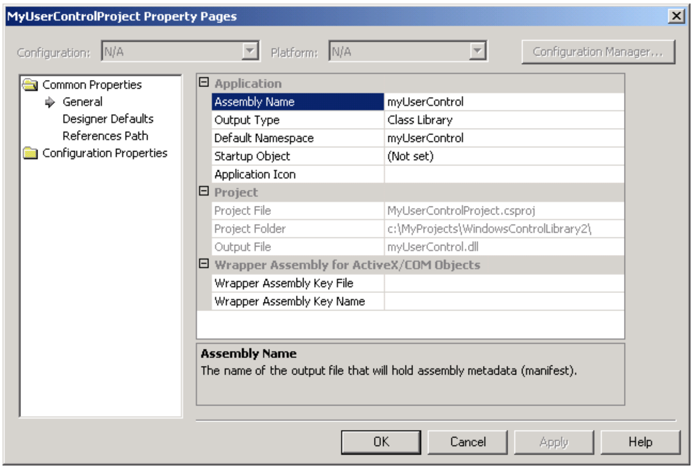


My Custom Control

☐ Bold

☐ Italic

Fig. 13.43 Custom-control creation.



MyUserControlProject Property Pages

Configuration: N/A   Platform: N/A   Configuration Manager...

- Common Properties
  - General
  - Designer Defaults
  - References Path
- Configuration Properties

Application

Assembly Name	myUserControl
Output Type	Class Library
Default Namespace	myUserControl
Startup Object	(Not set)
Application Icon	

Project

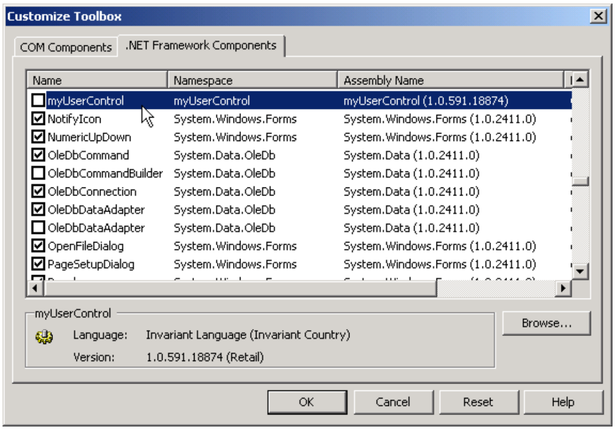
Project File	MyUserControlProject.csproj
Project Folder	c:\MyProjects\WindowsControlLibrary2\
Output File	myUserControl.dll

Wrapper Assembly for ActiveX/COM Objects

Wrapper Assembly Key File	
Wrapper Assembly Key Name	

Assembly Name
The name of the output file that will hold assembly metadata (manifest).

Fig. 13.44 Project properties dialog.



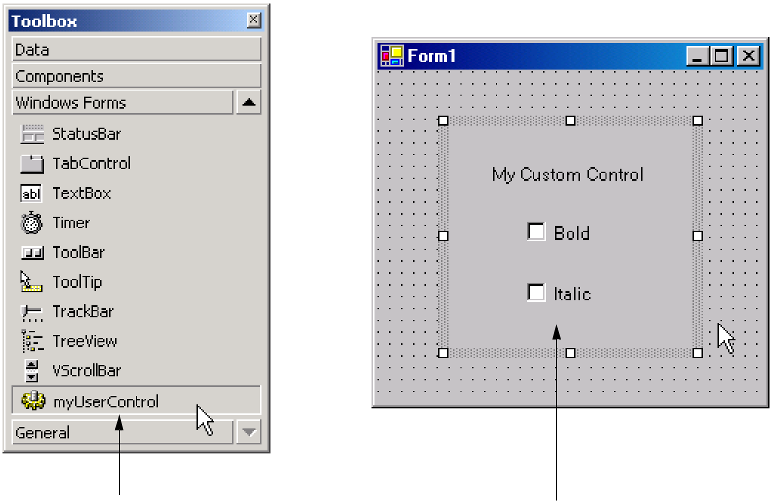
Customize Toolbox

COM Components | .NET Framework Components

Name	Namespace	Assembly Name
☐ myUserControl	myUserControl	myUserControl (1.0.591.18874)
☑ NotifyIcon	System.Windows.Forms	System.Windows.Forms (1.0.2411.0)
☑ NumericUpDown	System.Windows.Forms	System.Windows.Forms (1.0.2411.0)
☑ OleDbCommand	System.Data.OleDb	System.Data (1.0.2411.0)
☐ OleDbCommandBuilder	System.Data.OleDb	System.Data (1.0.2411.0)
☑ OleDbConnection	System.Data.OleDb	System.Data (1.0.2411.0)
☑ OleDbDataAdapter	System.Data.OleDb	System.Data (1.0.2411.0)
☐ OleDbDataAdapter	System.Data.OleDb	System.Data (1.0.2411.0)
☑ OpenFileDialog	System.Windows.Forms	System.Windows.Forms (1.0.2411.0)
☑ PageSetupDialog	System.Windows.Forms	System.Windows.Forms (1.0.2411.0)

myUserControl
Language: Invariant Language (Invariant Country)
Version: 1.0.591.18874 (Retail)

Fig. 13.45 Custom control added to the **ToolBox**.



New **ToolBox** icon

Newly inserted control

Fig. 13.46 Custom control added to a `Form`.

Many of today's most successful commercial programs provide GUIs that are easy to use and manipulate. Because of this demand for user-friendly GUIs, the ability to design sophisticated GUIs is an essential programming skill. Fortunately, Visual Studio .NET provides an IDE that makes GUI development quick and easy. In the last two chapters, we have presented the basic techniques required to add various GUI components to a program.The next chapter will explore a more behind-the-scenes topic, *multithreading*. In many programming languages, the programmer can create multiple *threads*, enabling several processes to occur at once. By learning to create and manage multithreading in C#, readers will begin their study of a more robust type of software.

SUMMARY

- Menus used to provide groups of related commands for Windows applications. Menus are an integral part of GUIs, because they enable user–application interaction without unnecessarily "cluttering" the GUI.
- Window's top-level menus appear on the left of the screen—any submenus or menu items are indented. All menu items can have *Alt* key shortcuts (also called access shortcuts or hot keys).
- Non-top-level menus can have shortcut keys (combinations of *Ctrl*, *Shift*, *Alt*, function keys *F1*, *F2*, letter keys etc.).
- To create a menu, open the **Toolbox**, and drag a `MainMenu` control onto the form.
- To add entries to the menu, click the **Type Here** textbox, and type the text that should appear in the menu. Remove a menu item by selecting it with the mouse and pressing the *Delete* key.
- Menus generate a `Click` event when selected.
- Use the `Xor` (exclusive OR) operator to toggle single bits, such as those representing the bold and italic styles.
- The `LinkLabel` control is used to display links to other objects, such as files or Web pages. The links can change color to reflect whether each link is new, visited or active.

- When clicked, a `LinkLabel` generate a `LinkClicked` event.
- Method `Start` of class `Process` (namespace `System.Diagnostics`) can begin a new application. This method requires either the file to open (a `String`) or the application to run and the command-line arguments (two `String`s).
- The `ListBox` control allows the user to view and select multiple items from a list.
- The `CheckedListBox` control extends a `ListBox` by accompanying each item in the list with a checkbox. This allows multiple items to be selected with no logical restriction.
- The `SelectionMode` property determines how many items in a `CheckedListBox` can be selected.
- The `SelectedIndexChanged` event occurs when the user selects a new item in a `CheckedListBox`.
- `CheckBox`’s property `Items` returns all the objects in the list as a collection. Property `SelectedItem` returns the currently selected item. `SelectedIndex` returns the index of the selected item.
- Method `GetSelected` takes an index and returns `True` if the corresponding item is selected.
- Add items visually by examining the `Items` collection in the **Properties** window. Clicking the ellipsis brings up the **String Collection Editor**, in which we can type the items to add.
- `CheckedListBox`es imply that multiple items can be selected—the `SelectionMode` property can only have values `None` or `One`. `One` allows multiple selection.
- Event `ItemCheck` is generated whenever a `CheckedListBox` item is about to change.
- The `ComboBox` control combines `TextBox` features with a drop-down list. The user can either select an option from the list or type one in (if allowed by the programmer). If the number of elements exceeds the maximum that can be displayed in the drop-down list, a scrollbar appears.
- Property `DropDownStyle` determines the type of `ComboBox`.
- The `ComboBox` control has properties `Items` (a collection), `SelectedItem` and `SelectedIndex`, which are similar to the corresponding properties in `ListBox`.
- When the selected item changes, event `SelectedIndexChanged` is generated.
- A `Graphics` object allows a pen or brush to draw on a component, via one of several `Graphics` methods.
- The `TreeView` control can display nodes hierarchically on a tree.
- A node is an element that contains a value and references to other nodes.
- A parent node contains child nodes, and the child nodes can be parents themselves.
- A tree is a collection of nodes, usually organized in some manner. The first parent node of a tree is often called the root node.
- Each node has a `Nodes` collection, which contains a list of the `Node`’s children.
- To add nodes to the `TreeView` visually, click the ellipsis by the `Nodes` property in the **Properties** window. This opens the **TreeNode Editor**, where there are buttons to create a root and to add, delete and rename nodes.
- Method `GetDirectories` takes a `String` (the current directory) and returns an array of `String`s (the subdirectories).
- The `ListView` control is similar to a `ListBox`—it displays a list from which the user can select one or more items. However, a `ListView` can display icons alongside the list items in a variety of ways.

- To display images, the programmer must use an **ImageList** component. Create one by dragging it onto the form from the **ToolBox**. Click the **Images** collection in the **Properties** window to display the **Image Collection Editor**.
- Class **DirectoryInfo** (namespace **System.IO**) allows us to browse or manipulate the directory structure easily. Method **GetDirectories** returns an array of **DirectoryInfo** objects containing the subdirectories of the current directory. Method **GetFiles** returns an array of class **FileInfo** objects containing the files in the current directory.
- The **TabControl** control creates tabbed windows. This allows the programmer to provide large quantities of information while saving screen space.
- **TabControl**s contain **TabPage** objects, which can contain controls.
- To add **TabPage**s in the Visual Studio .NET designer, right-click the **TabControl**, and select **Add Tab**.
- Each **TabPage** generates its own **Click** event when its tab is clicked. Events for controls inside the **TabPage** are still handled by the form.
- Single-document-interface (SDI) applications can support only one open window or document at a time. Multiple-document-interface (MDI) programs allows users to edit multiple documents at a time.
- Each window inside an MDI application is called a child window, and the application window is called the parent window.
- To create an MDI form, set the form's **IsMDIContainer** property to **True**.
- The parent and child windows of an application can have different menus, which are merged (combined) whenever a child window is selected.
- Class **MenuItem** property **MdiList** (a boolean) allows a menu item to contain a list of open child windows.
- The child windows in an MDI application can be arranged by calling method **LayoutMdi** of the parent form.
- The .NET Framework allows the programmer to create customized controls. The most basic way to create a customized control is to derive a class from an existing Windows Forms control. If we inherit from an existing Windows Forms control, we can add to its appearance, but not redesign it. To create a new control composed of existing controls, use class **UserControl**. To create a new control from the ground up, inherit from class **Control**.
- **Timer**s are invisible components that reside on a form and generate **Tick** events at a set interval.
- We create a **UserControl** class for the project by selecting **Project**, then **Add User Control...**. We can treat this control like a Windows Form, meaning that we can add controls, using the **ToolBox**, and set properties, using the **Properties** window.
- Structure **DateTime** (namespace **System**) contains member **Now**, which is the current time.

TERMINOLOGY

& (menu access shortcut)
access shortcut
Activation property of class **ListView**
ActiveLinkColor property of class **LinkLabel**
ActiveMdiChild property of class **Form**
Add member of enumeration **MenuMerge**
Add method of class **TreeNodeCollection**
Add Tab menu item
Add User Control... option in Visual Studio
Add Windows Form... option in Visual Studio
adding controls to **ToolBox**
AfterSelect event of class **TreeView**
ArrangeIcons value in **LayoutMdi** enumeration
boundary of a control

Cascade value in **LayoutMdi** enumeration
CheckBoxes property of class **ListView**
CheckBoxes property of class **TreeView**
Checked property of class **MenuItem**
Checked property of class **TreeNode**
CheckedIndices property of class **CheckedListBox**
CheckedItems property of class **CheckedListBox**
CheckedListBox class
child node
child window
child window maximized
child window minimized
Clear method of class **TreeNodeCollection**
Click event of class **MenuItem**
ClipRectangle property of class **PaintEventArgs**
Collapse method of class **TreeNode**
collapsing a node
ComboBox class
control boundary
Control class
CurrentValue event of class **CheckedListBox**
custom control
custom control being adding to **ToolBox**
Customize Toolbox... option in Visual Studio
DateTime structure
DirectoryInfo class
displaying files and folders in a **ListView**
draw on a control
DrawEllipse method of class **Graphics**
DrawPie method of class **Graphics**
DrawRectangle method of class **Graphics**
drop-down list
DropDown style for **ComboBox**
DropDownList style for **ComboBox**
DropDownStyle property of class **ComboBox**
events at an interval
Exit method of class **Application**
Expand method of class **TreeNode**
ExpandAll method of class **TreeNode**
expanding a node
FillEllipse method of class **Graphics**
FillPie method of class **Graphics**
FillRectange method of class **Graphics**
FirstNode property of class **TreeNode**
FullName property
FullPath property of class **TreeNode**
GetDirectories method of class **Directory**
GetDirectories method of class **DirectoryInfo**
GetFiles method of class **DirectoryInfo**
GetItemChecked method of class **CheckedListBox**
GetNodeCount method of class **TreeNode**
GetSelected method of class **ListBox**
Graphics class
Graphics property of class **PaintEventArgs**
hot key
Image Collection Editor
ImageIndex property of class **ListViewItem**
ImageIndex property of class **TreeNode**
ImageList class
ImageList collection
ImageList property of class **TabControl**
ImageList property of class **TreeView**
Index event of class **CheckedListBox**
Index property of class **MenuItem**
inherit from a Windows Form control
Insert Separator option
Interval property of class **Timer**
IsMdiChild property of class **Form**
IsMdiContainer property of class **Form**
ItemActivate event of class **ListView**
ItemCheck event of class **CheckedListBox**
ItemCheckEventArgs event of class **CheckedListBox**
Items property of class **ComboBox**
Items property of class **ListBox**
Items property of class **ListView**
ItemSize property of class **TabControl**
LargeImageList property of class **ListView**
LastNode property of class **TreeNode**
LayoutMdi enumeration
LayoutMdi method of class **Form**
LinkArea property of class **LinkLabel**
LinkBehavior property of class **LinkLabel**
LinkClicked event of class **LinkLabel**
LinkColor property of class **LinkLabel**
LinkLabel class
Links property of class **LinkLabel**
LinkVisited property of class **LinkLabel**
ListBox class

ListView class
Main method
MainMenu class
MaxDropDownItems property of class **ComboBox**
MDI form
MDI parent-window class
MDI title bar
MdiChildActivate event of class **Form**
MdiChildren property of class **Form**
MdiList property of class **MenuItem**
MdiParent property of class **Form**
menu
menu-access shortcut
Menu Designer in Visual Studio .NET
menu item
menu, expanded and checked
MenuItem class
MenuItems property of class **MainMenu**
MenuItems property of class **MenuItem**
MenuMerge enumeration
MergeItems member of enumeration **MenuMerge**
MergeOrder property of class **MenuItem**
MergeType property of class **MenuItem**
More Windows... option in Visual Studio .NET
MultiColumn property of class **ListBox**
MultiExtended value of **SelectionMode**
MultiLine property of class **TabControl**
multiple-document interface (MDI)
MultiSelect property of class **ListView**
MultiSimple value of **SelectionMode**
Name property of class **DirectoryInfo**
Name property of class **FileInfo**
NewValue event of class **CheckedListBox**
NextNode property of class **TreeNode**
Nodes property of class **TreeNode**
Nodes property of class **TreeView**
None value of **SelectionMode**
Now property of structure **DateTime**
One value of **SelectionMode**
OnPaint method
opening a file in Windows
output directory
PaintEventArgs class
parent menu
parent node
parent window
PictureBox class
PrevNode property of class **TreeNode**
Process class
project properties dialog
project, Windows control library
radio buttons, using with **TabPage**
RadioCheck property of class **MenuItem**
Remove member of enumeration **MenuMerge**
Remove method of class **TreeNodeCollection**
Replace member of enumeration **MenuMerge**
RightToLeft property of class **MainMenu**
root node
SelectedImageIndex property of class **TreeNode**
SelectedIndex property of class **ComboBox**
SelectedIndex property of class **ListBox**
SelectedIndex property of class **TabControl**
SelectedIndexChanged event of class **ComboBox**
SelectedIndexChanged event of class **ListBox**
SelectedIndexChanged event of class **TabControl**
SelectedIndices property of class **ListBox**
SelectedItem property of class **ComboBox**
SelectedItem property of class **ListBox**
SelectedItems property of class **ListBox**
SelectedItems property of class **ListView**
SelectedNode property of class **TreeView**
SelectedTab property of class **TabControl**
SelectionMode enumeration
SelectionMode property of class **CheckedListBox**
SelectionMode property of class **ListBox**
separator bar
separator, menu
shortcut key
Shortcut property of class **MenuItem**
Show method of class **Form**
ShowShortcut property of class **MenuItem**
Simple style for **ComboBox**
single-document interface (SDI)
SmallImageList property of class **ListView**
Solution Explorer in Visual Studio .NET
Sorted property of class **ComboBox**
Sorted property of class **ListBox**
Start method of class **Process**
String Collection Editor in Visual Studio .NET

submenu
`TabControl` class
`TabControl`, adding a **`TabPage`**
`TabCount` property of class **`TabControl`**
`TabPage` class
`TabPage`, add to **`TabControl`**
`TabPage`, using radio buttons
`TabPages` property of class **`TabControl`**
`Text` property of class **`LinkLabel`**
`Text` property of class **`MenuItem`**
`Text` property of class **`TreeNode`**
`Tick` event of class **`Timer`**
`TileHorizontal` value in **`LayoutMdi`** enumeration
`TileVertical` value in **`LayoutMdi`** enumeration
ToolBox customization
tree
`TreeNode` class
TreeNode Editor in VS .NET
`TreeView` class
`UseMnemonic` property of class **`LinkLabel`**
user-defined control
`UserControl` class
`View` property of class **`ListView`**
`VisitedLinkColor` property of class **`LinkLabel`**

SELF-REVIEW EXERCISES

13.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Menus provide groups of related classes.
b) Menu items can display radio buttons, checkmarks and access shortcuts.
c) The **`ListBox`** control allows only single selection (like a radio button), whereas the **`CheckedListBox`** allows multiple selection (like a check box).
d) The **`ComboBox`** control has a drop-down list.
e) Deleting a parent node in a **`TreeView`** control deletes its child nodes.
f) The user can select only one item in a **`ListView`** control.
g) A **`TabPage`** can act as a logical group for radio buttons.
h) In general, Multiple Document Interface (MDI) windows are used with simple applications.
i) An MDI child window can have MDI children.
j) MDI child windows cannot be maximized (enlarged) inside their parent.
k) There are two basic ways to create a customized control.

13.2 Fill in the blanks in each of the following statements:

a) Method __________ of class **`Process`** can open files and Web pages, much as can the **Run** menu in Windows.
b) If more elements appear in a **`ComboBox`** than can fit, a __________ appears.
c) The top-level node in a **`TreeView`** is the __________ node.
d) An **`ImageList`** is used to display icons in a __________.
e) The **`MergeOrder`** and **`MergeType`** properties determine how __________ merge.
f) The __________ property allows a menu to display a list of active child windows.
g) An important feature of the **`ListView`** control is the ability to display __________.
h) Class __________ allows the programmer to combine several controls into a single, custom control.
i) The __________ saves space by layering **`TabPage`**s on top of each other.
j) The __________ window layout option makes all windows the same size and layers them so every title bar is visible (if possible).
k) __________ are typically used to display hyperlinks to other objects, files or Web pages.

ANSWERS TO SELF-REVIEW EXERCISES

13.1 a) False. Menus provide groups of related commands. b) True. c) False. Both controls can have single or multiple selection. d) True. e) True. f) False. The user can select one or more items.

g) True. h) False. MDI windows tend to be used with complex applications. i) False. Only an MDI parent window can have MDI children. An MDI parent window cannot be an MDI child. j) False. MDI child windows cannot be moved outside their parent window. k) False. There are three methods: 1) Derive from an existing control, 2) use a `UserControl` or 3) derive from `Control` and create a control from scratch.

13.2 a) `Start`. b) scrollbar. c) root. d) `ListView`. e) menus. f) `MdiList`. g) icons. h) `UserControl`. i) `TabControl`. j) `Cascade`. k) `LinkLabel`s.

EXERCISES

13.3 Write a program that displays the names of 15 states in a `ComboBox`. When an item is selected from the `ComboBox`, remove it.

13.4 Modify your solution to Exercise 13.3 to add a `ListBox`. When the user selects an item from the `ComboBox`, remove the item from the `ComboBox`, and add it to the `ListBox`. Your program should check to ensure that the `ComboBox` contains at least one item. If it does not, print a message in a message box, and terminate program execution.

13.5 Write a program that allows the user to enter strings in a `TextBox`. Each string input is added to a `ListBox`. As each string is added to the `ListBox`, ensure that the strings are in sorted order. Any sorting method may be used.

13.6 Create a file browser (similar to Windows Explorer) based on the programs in Fig. 13.7, Fig. 13.21 and Fig. 13.24. The file browser should have a `TreeView`, which allows the user to browse directories. There should also be a `ListView`, which displays the contents (all subdirectories and files) of the directory being browsed. Double-clicking a file in the `ListView` should open it, and double-clicking a directory in either the `ListView` or the `TreeView` should browse it. If a file or directory cannot be accessed, because of its permission settings, notify the user.

13.7 Create an MDI text editor. Each child window should contain a multiline `TextBox`. The MDI parent should have a **Format** menu, with submenus to control the size, font and color of the text in the active child window. Each submenu should have at least three options. In addition, the parent should have a **File** menu with menu items **New** (create a new child), **Close** (close the active child) and **Exit** (exit the application). The parent should have a **Window** menu to display a list of the open child windows and their layout options.

13.8 Create a `UserControl` called `LoginPasswordUserControl`. The `LoginPasswordUserControl` contains a `Label` (`loginLabel`) that displays `String "Login:"`, a `TextBox` (`loginTextBox`) where the user inputs a login name, a `Label` (`passwordLabel`) that displays the `String "Password:"` and finally, a `TextBox` (`passwordTextBox`) where a user inputs a password (don't forget to set property `PasswordChar` to `"*"` in the `TextBox`'s **Properties** window). `LoginPasswordUserControl` must provide public read-only properties `Login` and `Password` that allow an application to retrieve the user input from `loginTextBox` and `passwordTextBox`. The `UserControl` must be exported to an application that displays the values input by the user in `LoginPasswordUserControl`.

14

Multithreading

Objectives

- To understand the notion of multithreading.
- To appreciate how multithreading can improve program performance.
- To understand how to create, manage and destroy threads.
- To understand the life cycle of a thread.
- To understand thread synchronization.
- To understand thread priorities and scheduling.
- To understand the role of a **`ThreadPool`** in efficient multithreading.

The spider's touch, how exquisitely fine!
Feels at each thread, and lives along the line.
Alexander Pope

A person with one watch knows what time it is; a person with two watches is never sure.
Proverb

Learn to labor and to wait.
Henry Wadsworth Longfellow

The most general definition of beauty…Multeity in Unity.
Samuel Taylor Coleridge

Outline

14.1 Introduction
14.2 Thread States: Life Cycle of a Thread
14.3 Thread Priorities and Thread Scheduling
14.4 Thread Synchronization and Class `Monitor`
14.5 Producer/Consumer Relationship without Thread Synchronization
14.6 Producer/Consumer Relationship with Thread Synchronization
14.7 Producer/Consumer Relationship: Circular Buffer

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

14.1 Introduction

It would be nice if we could perform one action at a time and perform it well, but that is usually difficult to do. The human body performs a great variety of operations *in parallel*—or, as we will say throughout this chapter, *concurrently*. Respiration, blood circulation and digestion, for example, can occur concurrently. All the senses—sight, touch, smell, taste and hearing—can occur at once. Computers, too, perform operations concurrently. It is common for desktop personal computers to be compiling a program, sending a file to a printer and receiving electronic mail messages over a network concurrently.

Ironically, most programming languages do not enable programmers to specify concurrent activities. Rather, programming languages generally provide only a simple set of control structures that enable programmers to perform one action at a time, proceeding to the next action after the previous one has finished. Historically, the type of concurrency that computers perform today generally has been implemented as operating system "primitives" available only to highly experienced "systems programmers."

The Ada programming language, developed by the United States Department of Defense, made concurrency primitives widely available to defense contractors building military command-and-control systems. However, Ada has not been widely used in universities and commercial industry.

The .NET Framework Class Library makes concurrency primitives available to the applications programmer. The programmer specifies that applications contain "threads of execution," each thread designating a portion of a program that may execute concurrently with other threads—this capability is called *multithreading*. Multithreading is available to all .NET programming languages, including C#, Visual Basic and Visual C++.

Software Engineering Observation 14.1

The .NET Framework Class Library includes multithreading capabilities in namespace **`System.Threading`**. *This encourages the use of multithreading among a larger part of the applications-programming community.*

We discuss many applications of concurrent programming. When programs download large files, such as audio clips or video clips from the World Wide Web, users do not want to wait until an entire clip downloads before starting the playback. To solve this problem, we can put multiple threads to work—one thread downloads a clip, and another plays the clip. These activities, or *tasks*, then may proceed concurrently. To avoid choppy playback, we *synchronize* the threads so that the player thread does not begin until there is a sufficient amount of the clip in memory to keep the player thread busy.

Another example of multithreading is C#'s automatic *garbage collection*. C and C++ place with the programmer the responsibility of reclaiming dynamically allocated memory. C# provides a *garbage-collector thread* that reclaims dynamically allocated memory that is no longer needed.

Performance Tip 14.1

One of the reasons for the popularity of C and C++ over the years was that their memory-management techniques were more efficient than those of languages that used garbage collectors. In fact, memory management in C# often is faster than in C or C++.[1]

Good Programming Practice 14.1

Set an object reference to `null` *when the program no longer needs that object. This enables the garbage collector to determine at the earliest possible moment that the object can be garbage collected. If such an object has other references to it, that object cannot be collected.*

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel "trains of thought." To see why multithreading can be difficult to program and understand, try the following experiment: Open three books to page 1 and try reading the books concurrently. Read a few words from the first book, then read a few words from the second book, then read a few words from the third book, then loop back and read the next few words from the first book, etc. After this experiment, you will appreciate the challenges of multithreading—switching between books, reading briefly, remembering your place in each book, moving the book you are reading closer so you can see it, pushing books you are not reading aside—and amidst all this chaos, trying to comprehend the content of the books!

Performance Tip 14.2

A problem with single-threaded applications is that lengthy activities must complete before other activities can begin. In a multithreaded application, threads can share a processor (or set of processors), so that multiple tasks are performed in parallel.

14.2 Thread States: Life Cycle of a Thread

At any time, a thread is said to be in one of several *thread states* (illustrated in Fig. 14.1[2]). This section discusses these states and the transitions between states. Two classes critical for multithreaded applications are ***`Thread`*** and ***`Monitor`*** (`System.Threading` namespace). This section also discusses several methods of classes `Thread` and `Monitor` that cause state transitions.

---

[1] E. Schanzer, "Performance Considerations for Run-Time Technologies in the .NET Framework," August 2001 `<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftechs.asp>`.

[2] As this book went to publication, Microsoft changed the names of the *Started* and *Blocked* thread states to *Running* and *WaitSleepJoin*, respectively.

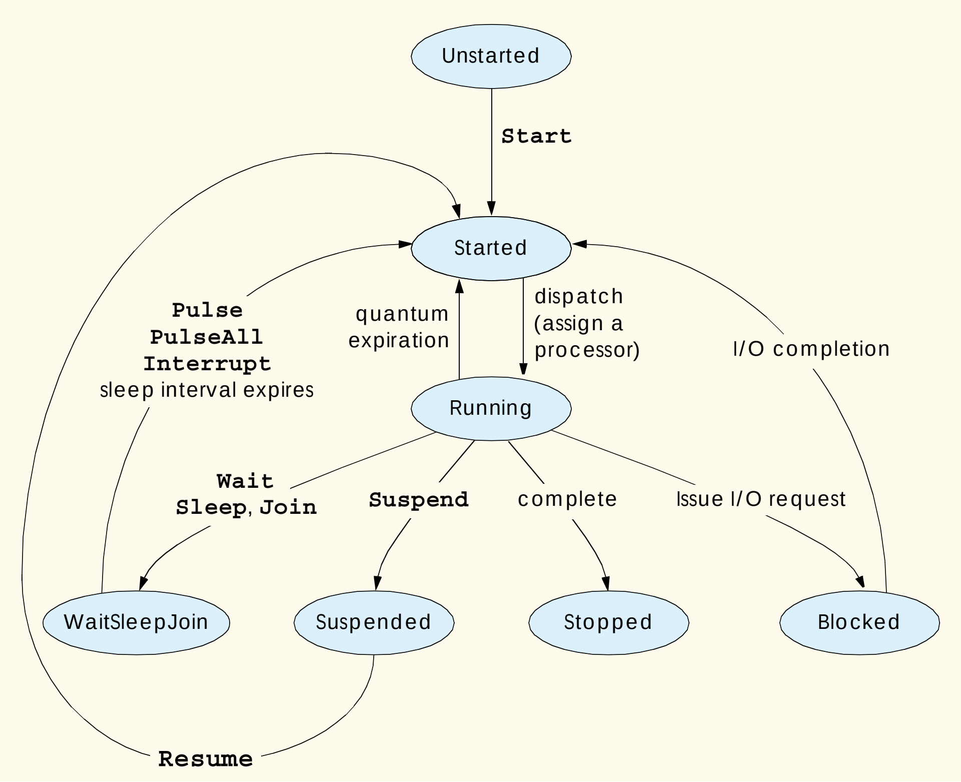

Unstarted
Start
Started
Pulse
PulseAll
Interrupt
sleep interval expires
quantum expiration
dispatch (assign a processor)
I/O completion
Running
Wait
Sleep, **Join**
Suspend
complete
Issue I/O request
WaitSleepJoin
Suspended
Stopped
Blocked
Resume

Fig. 14.1 Thread life cycle.

A new thread begins its lifecyle in the *Unstarted* state. The thread remains in the *Unstarted* state until the program calls **`Thread`** method ***`Start`***, which places the thread in the *Started* state (sometimes called the *Ready* or *Runnable* state) and immediately returns control to the calling thread. Then the thread that invoked **`Start`**, the newly *Started* thread and any other threads in the program execute concurrently.

The highest priority *Started* thread enters the *Running* state (i.e., begins executing) when the operating system assigns a processor to the thread (Section 14.3 discusses thread priorities). When a *Started* thread receives a processor for the first time and becomes a *Running* thread, the thread executes its ***`ThreadStart`*** delegate, which specifies the actions the thread will perform during its lifecyle. When a program creates a new **`Thread`**, the program specifies the **`Thread`**'s **`ThreadStart`** delegate as the argument to the **`Thread`** constructor. The **`ThreadStart`** delegate must be a method that returns **`void`** and takes no arguments.

A *Running* thread enters the *Stopped* (or *Dead*) state when its **`ThreadStart`** delegate terminates. Note that a program can force a thread into the *Stopped* state by calling **`Thread`** method ***`Abort`*** on the appropriate **`Thread`** object. Method **`Abort`** throws a ***`ThreadAbortException`*** in the thread, normally causing the thread to terminate. When a thread is in the *Stopped* state and there are no references to the thread object, the garbage collector can remove the thread object from memory.

A thread enters the *Blocked* state when the thread issues an input/output request. The operating system blocks the thread from executing until the operating system can complete the I/O for which the thread is waiting. At that point, the thread returns to the *Started* state, so it can resume execution. A *Blocked* thread cannot use a processor even if one is available.

There are three ways in which a *Running* thread enters the *WaitSleepJoin* state. If a thread encounters code that it cannot execute yet (normally because a condition is not satisfied), the thread can call ***`Monitor`*** method ***`Wait`*** to enter the *WaitSleepJoin* state. Once in this state, a thread returns to the *Started* state when another thread invokes **`Monitor`** method ***`Pulse`*** or ***`PulseAll`***. Method **`Pulse`** moves the next waiting thread back to the *Started* state. Method **`PulseAll`** moves all waiting threads back to the *Started* state.

A *Running* thread can call **`Thread`** method ***`Sleep`*** to enter the *WaitSleepJoin* state for a period of milliseconds specified as the argument to **`Sleep`**. A sleeping thread returns to the *Started* state when its designated sleep time expires. Sleeping threads cannot use a processor, even if one is available.

Any thread that enters the *WaitSleepJoin* state by calling **`Monitor`** method **`Wait`** or by calling **`Thread`** method **`Sleep`** also leaves the *WaitSleepJoin* state and returns to the *Started* state if the sleeping or waiting **`Thread`**'s ***`Interrupt`*** method is called by another thread in the program.

If a thread cannot continue executing (we will call this the dependent thread) unless another thread terminates, the dependent thread calls the other thread's ***`Join`*** method to "join" the two threads. When two threads are "joined," the dependent thread leaves the *WaitSleepJoin* state when the other thread finishes execution (enters the *Stopped* state).

If a *Running* **`Thread`**'s ***`Suspend`*** method is called, the *Running* thread enters the *Suspended* state. A *Suspended* thread returns to the *Started* state when another thread in the program invokes the Suspended thread's ***`Resume`*** method.

14.3 Thread Priorities and Thread Scheduling

Every thread has a priority in the range between **`ThreadPriority.Lowest`** to **`ThreadPriority.Highest`**. These two values come from the **`ThreadPriority`** enumeration (namespace **`System.Threading`**). The enumeration consists of the values **`Lowest`**, **`BelowNormal`**, **`Normal`**, **`AboveNormal`** and **`Highest`**. By default, each thread has priority **`Normal`**.

The Windows operating system supports a concept, called *timeslicing*, that enables threads of equal priority to share a processor. Without timeslicing, each thread in a set of equal-priority threads runs to completion (unless the thread leaves the *Running* state and enters the *WaitSleepJoin*, *Suspended* or *Blocked* state) before the thread's peers get a chance to execute. With timeslicing, each thread receives a brief burst of processor time, called a *quantum*, during which the thread can execute. At the completion of the quantum, even if the thread has not finished executing, the processor is taken away from that thread and given to the next thread of equal priority, if one is available.

The job of the thread scheduler is to keep the highest-priority thread running at all times and, if there is more than one highest-priority thread, to ensure that all such threads execute for a quantum in round-robin fashion. Figure 14.2 illustrates the multilevel priority queue for threads. In Fig. 14.2, assuming a single-processor computer, threads A and B each execute for a quantum in round-robin fashion until both threads complete execution. This means that A gets a quantum of time to run. Then B gets a quantum. Then A gets another quantum. Then

B gets another quantum. This continues until one thread completes. The processor then devotes all its power to the thread that remains (unless another thread of that priority is *Started*). Next, thread C runs to completion. Threads D, E and F each execute for a quantum in round-robin fashion until they all complete execution. This process continues until all threads run to completion. Note that, depending on the operating system, new higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such *indefinite postponement* often is referred to more colorfully as *starvation*.

A thread's priority can be adjusted with the ***`Priority`*** property, which accepts values from the **`ThreadPriority`** enumeration. If the argument is not one of the valid thread-priority constants, an **`ArgumentException`** occurs.

A thread executes until it dies, becomes *Blocked* for input/output (or some other reason), calls **`Sleep`**, calls **`Monitor`** method **`Wait`** or **`Join`**, is preempted by a thread of higher priority or has its quantum expire. A thread with a higher priority than the *Running* thread can become *Started* (and hence preempt the *Running* thread) if a sleeping thread wakes up, if I/O completes for a thread that *Blocked* for that I/O, if either **`Pulse`** or **`PulseAll`** is called on an object on which **`Wait`** was called, or if a thread to which the high-priority thread was **`Join`**ed completes.

Figure 14.3 demonstrates basic threading techniques, including the construction of a **`Thread`** object and using the **`Thread`** class's **`static`** method **`Sleep`**. The program creates three threads of execution, each with the default priority **`Normal`**. Each thread displays a message indicating that it is going to sleep for a random interval of from 0 to 5000 milliseconds, then goes to sleep. When each thread awakens, the thread displays its name, indicates that it is done sleeping, terminates and enters the *Stopped* state. You will see that method **`Main`** (i.e., the ***`Main`*** *thread of execution*) terminates before the application terminates. The program consists of two classes—**`ThreadTester`** (lines 8–41), which creates the three threads, and **`MessagePrinter`** (lines 44–73), which defines a **`Print`** method containing the actions each thread will perform.

Objects of class **`MessagePrinter`** (lines 44–73) control the lifecycle of each of the three threads class **`ThreadTester`**'s **`Main`** method creates. Class **`MessagePrinter`** consists of instance variable **`sleepTime`** (line 46), **`static`** variable **`random`** (line 47), a constructor (lines 50–54) and a **`Print`** method (lines 57–71). Variable **`sleepTime`** stores a random integer value chosen when a new **`MessagePrinter`** object's constructor is called. Each thread controlled by a **`MessagePrinter`** object sleeps for the amount of time specified by the corresponding **`MessagePrinter`** object's **`sleepTime`**

The **`MessagePrinter`** constructor (lines 50–54) initializes **`sleepTime`** to a random integer from 0 up to, but not including, 5001 (i.e., from 0 to 5000).

Method **`Print`** begins by obtaining a reference to the currently executing thread (line 60) via class **`Thread`**'s **`static`** property ***`CurrentThread`***. The currently executing thread is the one that invokes method **`Print`**. Next, lines 63–64 display a message indicating the name of the currently executing thread and stating that the thread is going to sleep for a certain number of milliseconds. Note that line 64 uses the currently executing thread's **`Name`** property to obtain the thread's name (set in method **`Main`** when each thread is created). Line 66 invokes **`static`** **`Thread`** method **`Sleep`** to place the thread into the *WaitSleepJoin* state. At this point, the thread loses the processor and the system allows another thread to execute. When the thread awakens, it reenters the *Started* state again until the system assigns a processor to the thread. When the **`MessagePrinter`** object enters the

Running state again, line 69 outputs the thread's name in a message that indicates the thread is done sleeping, and method **`Print`** terminates.

Class **`ThreadTester`**'s **`Main`** method (lines 10–39) creates three objects of class **`MessagePrinter`**, at lines 14, 19 and 24, respectively. Lines 15–16, 20–21 and 25–26 create and initialize three **`Thread`** objects. Lines 17, 22 and 27 set each **`Thread`**'s **`Name`** property, which we use for output purposes. Note that each **`Thread`**'s constructor receives a **`ThreadStart`** delegate as an argument. Remember that a **`ThreadStart`** delegate specifies the actions a thread performs during its lifecyle. Line 16 specifies that the delegate for **`thread1`** will be method **`Print`** of the object to which **`printer1`** refers. When **`thread1`** enters the *Running* state for the first time, **`thread1`** will invoke **`printer1`**'s **`Print`** method to perform the tasks specified in method **`Print`**'s body. Thus, **`thread1`** will print its name, display the amount of time for which it will go to sleep, sleep for that amount of time, wake up and display a message indicating that the thread is done sleeping. At that point method **`Print`** will terminate. A thread completes its task when the method specified by a **`Thread`**'s **`ThreadStart`** delegate terminates, placing the thread in the *Stopped* state. When **`thread2`** and **`thread3`** enter the *Running* state for the first time, they invoke the **`Print`** methods of **`printer2`** and **`printer3`**, respectively. Threads **`thread2`** and **`thread3`** perform the same tasks as **`thread1`** by executing the **`Print`** methods of the objects to which **`printer2`** and **`printer3`** refer (each of which has its own randomly chosen sleep time).

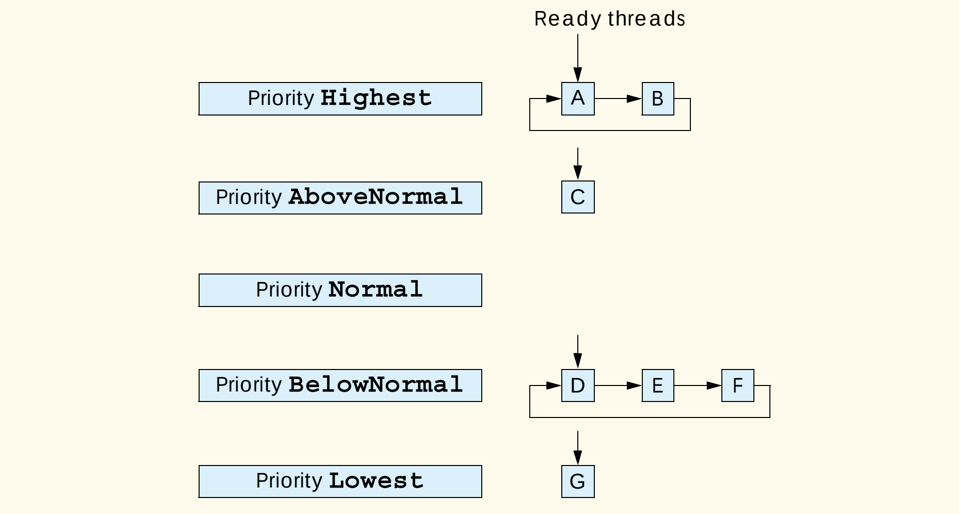


Fig. 14.2 Thread-priority scheduling.

```
1  // Fig. 14.3: ThreadTester.cs
2  // Multiple threads printing at different intervals.
3
```

Fig. 14.3 Threads sleeping and printing. (Part 1 of 3.)

```
using System;
using System.Threading;

// class ThreadTester demonstrates basic threading concepts
class ThreadTester
{
   static void Main( string[] args )
   {
      // Create and name each thread. Use MessagePrinter's
      // Print method as argument to ThreadStart delegate.
      MessagePrinter printer1 = new MessagePrinter();
      Thread thread1 =
         new Thread ( new ThreadStart( printer1.Print ) );
      thread1.Name = "thread1";

      MessagePrinter printer2 = new MessagePrinter();
      Thread thread2 =
         new Thread ( new ThreadStart( printer2.Print ) );
      thread2.Name = "thread2";

      MessagePrinter printer3 = new MessagePrinter();
      Thread thread3 =
         new Thread ( new ThreadStart( printer3.Print  ) );
      thread3.Name = "thread3";

      Console.WriteLine( "Starting threads" );

      // call each thread's Start method to place each
      // thread in Started state
      thread1.Start();
      thread2.Start();
      thread3.Start();

      Console.WriteLine( "Threads started\n" );

   } // end method Main

} // end class ThreadTester

// Print method of this class used to control threads
class MessagePrinter
{
   private int sleepTime;
   private static Random random = new Random();

   // constructor to initialize a MessagePrinter object
   public MessagePrinter()
   {
      // pick random sleep time between 0 and 5 seconds
      sleepTime = random.Next( 5001 );
   }

```

Fig. 14.3 Threads sleeping and printing. (Part 2 of 3.)

```
56       // method Print controls thread that prints messages
57       public void Print()
58       {
59          // obtain reference to currently executing thread
60          Thread current = Thread.CurrentThread;
61
62          // put thread to sleep for sleepTime amount of time
63          Console.WriteLine(
64             current.Name + " going to sleep for " + sleepTime );
65
66          Thread.Sleep ( sleepTime );
67
68          // print thread name
69          Console.WriteLine( current.Name + " done sleeping" );
70
71       } // end method Print
72
73    } // end class MessagePrinter
```

```
Starting threads
Threads started

thread1 going to sleep for 1977
thread2 going to sleep for 4513
thread3 going to sleep for 1261
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

```
Starting threads
Threads started

thread1 going to sleep for 1466
thread2 going to sleep for 4245
thread3 going to sleep for 1929
thread1 done sleeping
thread3 done sleeping
thread2 done sleeping
```

Fig. 14.3 Threads sleeping and printing. (Part 3 of 3.)

Testing and Debugging Tip 14.1

Naming threads helps in the debugging of a multithreaded program. Visual Studio .NET's debugger provides a **Threads** *window that displays the name of each thread and enables you to view the execution of any thread in the program.*

Lines 33–35 invoke each `Thread`'s `Start` method to place the threads in the *Started* state (sometimes called *launching a thread*). Method `Start` returns immediately from each invocation, then line 37 outputs a message indicating that the threads were started, and the `Main` thread of execution terminates. The program itself does not terminate, however, because there are still threads that are alive (i.e., the threads were *Started* and have not reached the *Stopped* state yet). The program will not terminate until its last thread dies. When the system assigns a processor to a thread, the thread enters the *Running* state and calls the method specified by the thread's **`ThreadStart`** delegate. In this program, each thread invokes method **`Print`** of the appropriate **`MessagePrinter`** object to perform the tasks discussed previously.

Note that the sample outputs for this program show each thread and the thread's sleep time as the thread goes to sleep. The thread with the shortest sleep time normally awakens first, then indicates that it is done sleeping and terminates. In Section 14.7, we discuss multithreading issues that could prevent the thread with the shortest sleep time from awakening first.

14.4 Thread Synchronization and Class `Monitor`

Often, multiple threads of execution manipulate shared data. If threads with access to shared data simply read that data, then there is no need to prevent the data from being accessed by more than one thread at a time. However, when multiple threads share data and that data is modified by one or more of those threads, then indeterminate results may occur. If one thread is in the process of updating the data and another thread tries to update it too, the data will reflect the update that occurs second. If the data is an array or other data structure in which the threads could update separate parts of the data concurrently, it is possible that part of the data will reflect the information from one thread while another part of the data will reflect information from a different thread. When this happens, the program has difficulty determining when the data has been updated properly.

The problem can be solved by giving one thread at a time exclusive access to code that manipulates the shared data. During that time, other threads desiring to manipulate the data should be kept waiting. When the thread with exclusive access to the data completes its manipulation of the data, one of the threads waiting to manipulate the data should be allowed to proceed. In this fashion, each thread accessing the shared data excludes all other threads from doing so simultaneously. This is called *mutual exclusion* or *thread synchronization*.

C# uses the .NET Framework's monitors[3] to perform synchronization. Class **`Monitor`** provides the methods for *locking objects* to implement synchronized access to shared data. Locking an object means that only one thread can access that object at a time. When a thread wishes to acquire exclusive control over an object, the thread invokes **`Monitor`** method ***`Enter`*** to acquire the lock on that data object. Each object has a *SyncBlock* that maintains the state of that object's lock. Methods of class **`Monitor`** use the data in an object's *SyncBlock* to determine the state of the lock for that object. After acquiring the lock for an object, a thread can manipulate that object's data. While the object is locked, all other threads attempting to acquire the lock on that object are blocked (i.e., they enter the *Blocked* state) from acquiring the lock. When the thread that locked the shared object no longer requires the lock, that thread invokes **`Monitor`** method ***`Exit`*** to release the lock. This updates the *SyncBlock* of the shared object to indicate that the lock for the object is available again. At this point, if there is a thread that was previously blocked from acquiring the lock on the shared object, that thread acquires the lock to begin its processing of the object. If all threads with access to an object attempt to acquire the object's lock before manipulating the object, only one thread at a time will be allowed to manipulate the object. This helps ensure the integrity of the data.

---

[3] Hoare, C. A. R. Monitors: An Operating System Structuring Concept, *Communications of the ACM*. Vol. 17, No. 10, October 1974: 549–557. *Corrigendum, Communications of the ACM*. Vol. 18, No. 2, February 1975: 95.

Common Programming Error 14.1

Make sure that all code that updates a shared object locks the object before doing so. Otherwise a thread calling a method that does not lock the object can make the object unstable even when another thread has acquired the lock for the object.

Common Programming Error 14.2

Deadlock occurs when a waiting thread (let us call this thread1*) cannot proceed because it is waiting for another thread (let us call this* thread2*) to proceed. Similarly,* thread2 *cannot proceed because it is waiting for* thread1 *to proceed. The two threads are waiting for each other; therefore, the actions that would enable each thread to continue execution never occur.*

C# provides another means of manipulating an object's lock—keyword ***`lock`***. Placing `lock` before a block of code (designated with braces) as in

```
lock ( objectReference )
{
   // code that requires synchronization goes here
}
```

obtains the lock on the object to which the *objectReference* in parentheses refers. The *objectReference* is the same reference that normally would be passed to `Monitor` methods `Enter`, `Exit`, `Pulse` and `PulseAll`. When a `lock` block terminates for any reason, C# releases the lock on the object to which the *objectReference* refers. We explain `lock` further in Section 14.7.

If a thread determines that it cannot perform its task on a locked object, the thread can call `Monitor` method `Wait` and pass as an argument the object on which the thread will wait until the thread can perform its task. Calling method `Monitor.Wait` from a thread releases the lock the thread has on the object `Wait` receives as an argument and places that thread into the *WaitSleepJoin* state for that object. A thread in the *WaitSleepJoin* state for an object leaves the *WaitSleepJoin* state when a separate thread invokes `Monitor` method `Pulse` or `PulseAll` with the object as an argument. Method `Pulse` transitions the object's first waiting thread from the *WaitSleepJoin* state to the *Started* state. Method `PulseAll` transitions all threads in the object's *WaitSleepJoin* state to the *Started* state. The transition to the *Started* state enables the thread (or threads) to get ready to continue executing.

There is a difference between threads waiting to acquire the lock for an object and threads waiting in an object's *WaitSleepJoin* state: The threads called `Monitor` method `Wait` with the object as an argument. Threads that are waiting to acquire the lock enter the *Blocked* state and wait there until the object's lock becomes available. Then, one of the blocked threads can acquire the object's lock.

`Monitor` methods `Enter`, `Exit`, `Wait`, `Pulse` and `PulseAll` all take a reference to an object—usually the keyword `this`—as their argument.

Common Programming Error 14.3

A thread in the WaitSleepJoin *state cannot reenter the* Started *state to continue execution until a separate thread invokes* **`Monitor`** *method* **`Pulse`** *or* **`PulseAll`** *with the appropriate object as an argument. If this does not occur, the waiting thread will wait forever and so can cause deadlock.*

Testing and Debugging Tip 14.2

When multiple threads manipulate a shared object, using monitors, ensure that, if one thread calls **`Monitor`** *method* **`Wait`** *to enter the* WaitSleepJoin *state for the shared object, a separate thread eventually will call* **`Monitor`** *method* **`Pulse`** *to transition the thread waiting on the shared object back to the* Started *state. If multiple threads may be waiting for the shared object, a separate thread can call* **`Monitor`** *method* **`PulseAll`** *as a safeguard to ensure that all waiting threads have another opportunity to perform their tasks.*

Performance Tip 14.3

Synchronization to achieve correctness in multithreaded programs can make programs run more slowly, as a result of monitor overhead and the frequent transitioning of threads among the Running, WaitSleepJoin *and* Started *states. There is not much to say, however, for highly efficient, incorrect multithreaded programs!*

14.5 Producer/Consumer Relationship without Thread Synchronization

In a *producer/consumer relationship*, the *producer* portion of an application generates data and the *consumer* portion of an application uses that data. In a multithreaded producer/consumer relationship, a *producer thread* calls a *produce method* to generate data and place it into a shared region of memory, called a *buffer*. A *consumer thread* calls a *consume method* to read that data. If the producer waiting to put the next data into the buffer determines that the consumer has not yet read the previous data from the buffer, the producer thread should call **`Wait`**; otherwise, the consumer never sees the previous data and that data is lost to that application. When the consumer thread reads the message, it should call **`Pulse`** to allow a waiting producer to proceed. If a consumer thread finds the buffer empty or finds that the previous data has already been read, the consumer should call **`Wait`**; otherwise, the consumer might read “garbage” from the buffer or the consumer might process a previous data item more than once—each of these possibilities results in a logic error in the application. When the producer places the next data into the buffer, the producer should call **`Pulse`** to allow the consumer thread to proceed.

Let us consider how logic errors can arise if we do not synchronize access among multiple threads manipulating shared data. Consider a producer/consumer relationship in which a producer thread writes a sequence of numbers (we use 1–4) into a *shared buffer*—a memory location shared between multiple threads. The consumer thread reads this data from the shared buffer then displays the data. We display in the program’s output the values that the producer writes (produces) and that the consumer reads (consumes). Figure 14.4 demonstrates a producer and a consumer accessing a single shared cell (**`int`** variable **`buffer`**) of memory without any synchronization. Both the consumer and the producer threads access this single cell: The producer thread writes to the cell; the consumer thread reads from it. We would like each value the producer thread writes to the shared cell to be consumed exactly once by the consumer thread. However, the threads in this example are not synchronized. Therefore, data can be lost if the producer places new data into the slot before the consumer consumes the previous data. Also, data can be incorrectly repeated if the consumer consumes data again before the producer produces the next item. To show these possibilities, the consumer thread in the following example keeps a total of all the values it reads. The producer thread produces values from 1 to 4. If the consumer reads each value produced once and only once, the total would be 10. However, if you execute this program several times, you will see that the total is rarely, if ever, 10. Also, to emphasize our point, the producer and consumer threads in the example each sleep for random intervals of up to three seconds between performing their tasks. Thus, we do not know exactly when the producer thread will attempt to write a new value, nor do we know when the consumer thread will attempt to read a value.

The program consists of four classes—**`HoldIntegerUnsynchronized`** (lines 9–34), **`Producer`** (lines 37–70), **`Consumer`** (73–106) and **`SharedCell`** (109–144).

```
 1  // Fig. 14.4: Unsynchronized.cs
 2  // Showing multiple threads modifying a shared object without
 3  // synchronization.
 4
 5  using System;
 6  using System.Threading;
 7
 8  // this class represents a single shared int
 9  public class HoldIntegerUnsynchronized
10  {
11     // buffer shared by producer and consumer threads
12     private int buffer = -1;
13
14     // property Buffer
15     public int Buffer
16     {
17        get
18        {
19           Console.WriteLine( Thread.CurrentThread.Name +
20              " reads " + buffer );
21
22           return buffer;
23        }
24
25        set
26        {
27           Console.WriteLine( Thread.CurrentThread.Name +
28              " writes " + value );
29
30           buffer = value;
31        }
32
33     } // end property Buffer
34
35  } // end class HoldIntegerUnsynchronized
```

Fig. 14.4 Producer and consumer threads accessing a shared object without synchronization. (Part 1 of 4.)

```

// class Producer's Produce method controls a thread that
// stores values from 1 to 4 in sharedLocation
class Producer
{
   private HoldIntegerUnsynchronized sharedLocation;
   private Random randomSleepTime;

   // constructor
   public Producer(
      HoldIntegerUnsynchronized shared, Random random )
   {
      sharedLocation = shared;
      randomSleepTime = random;
   }

   // store values 1-4 in object sharedLocation
   public void Produce()
   {
      // sleep for random interval upto 3000 milliseconds
      // then set sharedLocation's Buffer property
      for ( int count = 1; count <= 4; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
         sharedLocation.Buffer = count;
      }

      Console.WriteLine( Thread.CurrentThread.Name +
         " done producing.\nTerminating " +
         Thread.CurrentThread.Name + "." );

   } // end method Produce

} // end class Producer

// class Consumer's Consume method controls a thread that
// loops four times and reads a value from sharedLocation
class Consumer
{
   private HoldIntegerUnsynchronized sharedLocation;
   private Random randomSleepTime;

   // constructor
   public Consumer(
      HoldIntegerUnsynchronized shared, Random random )
   {
      sharedLocation = shared;
      randomSleepTime = random;
   }

```

Fig. 14.4 Producer and consumer threads accessing a shared object without synchronization. (Part 2 of 4.)

```
   // read sharedLocation's value four times
   public void Consume()
   {
      int sum = 0;

      // sleep for random interval up to 3000 milliseconds
      // then add sharedLocation's Buffer property value
      // to sum
      for ( int count = 1; count <= 4; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
         sum += sharedLocation.Buffer;
      }

      Console.WriteLine( Thread.CurrentThread.Name +
         " read values totaling: " + sum +
         ".\nTerminating " + Thread.CurrentThread.Name + "." );

   } // end method Consume

} // end class Consumer

// this class creates producer and consumer threads
class SharedCell
{
   // create producer and consumer threads and start them
   static void Main( string[] args )
   {
      // create shared object used by threads
      HoldIntegerUnsynchronized holdInteger =
         new HoldIntegerUnsynchronized();

      // Random object used by each thread
      Random random = new Random();

      // create Producer and Consumer objects
      Producer producer =
         new Producer( holdInteger, random );

      Consumer consumer =
         new Consumer( holdInteger, random );

      // create threads for producer and consumer and set
      // delegates for each thread
      Thread producerThread =
         new Thread( new ThreadStart( producer.Produce ) );
      producerThread.Name = "Producer";

      Thread consumerThread =
         new Thread( new ThreadStart( consumer.Consume ) );
      consumerThread.Name = "Consumer";

```

Fig. 14.4 Producer and consumer threads accessing a shared object without synchronization. (Part 3 of 4.)

```
      // start each thread
      producerThread.Start();
      consumerThread.Start();

   } // end method Main

} // end class SharedCell
```

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
```

```
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
```

```
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
```

Fig. 14.4 Producer and consumer threads accessing a shared object without synchronization. (Part 4 of 4.)

Class **`HoldIntegerUnsynchronized`** (lines 9–35) consists of instance variable **`buffer`** (line 12) and property **`Buffer`** (lines 15–33), which provides **`get`** and **`set`** accessors. Property **`Buffer`**'s accessors do not synchronize access to instance variable **`buffer`**. Note that each accessor uses class **`Thread`**'s **`static`** property **`CurrentThread`** to obtain a reference to the currently executing thread, then uses that thread's property **`Name`** to obtain the thread's name.

Class **`Producer`** (lines 39–69) consists of instance variable **`sharedLocation`** (line 41), instance variable **`randomSleepTime`** (line 42), a constructor (lines 45–50) to initialize the instance variables and a **`Produce`** method (lines 53–67). The constructor initializes instance variable **`sharedLocation`** to refer to the **`HoldIntegerUnsynchronized`** object received from method **`Main`** as the argument **`shared`**. The producer thread in this program executes the tasks specified in method **`Produce`** of class class **`Producer`**. Method **`Produce`** contains a **`for`** structure (lines 57–61) that loops four times. Each iteration of the loop first invokes **`Thread`** method **`Sleep`** to place producer thread into the *WaitSleepJoin* state for a random time interval between 0 and 3 seconds. When the thread awakens, line 61 assigns the value of control variable **`count`** to the **`HoldIntegerUnsynchronized`** object's **`Buffer`** property, which causes the **`set`** accessor of **`HoldIntegerUnsynchronized`** to modify the **`buffer`** instance variable of the **`HoldIntegerUnsynchronized`** object. When the loop completes, lines 63–65 display a line of text in the console window indicating that the thread finished producing data and that the thread is terminating, then the **`Produce`** method terminates and so places the producer thread in the *Stopped* state.

Class **`Consumer`** (73–106) consists of instance variable **`sharedLocation`** (line 75), instance variable **`randomSleepTime`** (line 76), a constructor (lines 79–84) to initialize the instance variables and a **`Consume`** method (lines 87–104). The constructor initializes **`sharedLocation`** to refer to the **`HoldIntegerUnsynchronized`** received from **`Main`** as the argument **`shared`**. The consumer thread in this program performs the tasks specified in class **`Consumer`**'s **`Consume`** method. The method contains a **`for`** structure (lines 94–98) that loops four times. Each iteration of the loop invokes **`Thread`** method **`Sleep`** to put the consumer thread into the *WaitSleepJoin* state for a random time interval between 0 and 3 seconds. Next, line 97 gets the value of the **`HoldIntegerUnsynchronized`** object's **`Buffer`** property and adds the value to the variable **`sum`**. When the loop completes, lines 100–102 display a line in the console window indicating the sum of all values read, then the **`Consume`** method terminates, which places the consumer thread in the *Stopped* state.

Note: We use method **`Sleep`** in this example to emphasize the fact that, in multithreaded applications, it is unclear when each thread will perform its task and for how long it will perform that task when it has the processor. Normally, these thread-scheduling issues are the job of the computer's operating system. In this program, our thread's tasks are quite simple—for the producer, loop four times and perform an assignment statement; for the consumer, loop four times and add a value to variable **`sum`**. Without the **`Sleep`** method call, and if the producer executes first, the producer would complete its task before the consumer ever gets a chance to execute. If the consumer executes first, it would consume -1 four times, then terminate before the producer can produce the first real value.

Class **`SharedCell`**'s **`Main`** method (lines 112–142) instantiates a shared **`HoldIntegerUnsynchronized`** object (lines 115–116) and a **`Random`** object (line 119) for generating random sleep times and uses them as arguments to the constructors for the objects of classes **`Producer`** (lines 122–123) and **`Consumer`** (lines 125–126). The **`HoldIntegerUnsynchronized`** object contains the data that will be shared between the producer and consumer threads. Lines 130–132 create and name **`producerThread`**. The **`ThreadStart`** delegate for **`producerThread`** specifies that the thread will execute method **`Produce`** of object **`producer`**. Lines 134–136 create and name the **`consumerThread`**. The **`ThreadStart`** delegate for the **`consumerThread`** specifies that the thread will execute method **`Consume`** of object **`consumer`**. Finally, lines 139–140 place the two threads in the *Started* state by invoking each thread's **`Start`** method, then the **`Main`** thread terminates.

Ideally, we would like every value produced by the **`Producer`** object to be consumed exactly once by the **`Consumer`** object. However, when we study the first output of Fig. 14.4, we see that the consumer retrieved a value (-1) before the producer ever placed a value in the shared buffer and that the value 1 was consumed three times. The consumer finished executing before the producer had an opportunity to produce the values 2, 3 and 4. Therefore, those three values were lost. In the second output, we see that the value 1 was lost, because the values 1 and 2 were produced before the consumer thread could read the value 1. Also, the value 4 was consumed twice. The last sample output demonstrates that it is possible, with some luck, to get a proper output in which each value the producer produces is consumed once and only once by the consumer. This example clearly demonstrates that access to shared data by concurrent threads must be controlled carefully; otherwise, a program may produce incorrect results.

To solve the problems of lost data and data consumer more than once in the previous example, we will (in Fig. 14.5) synchronize access of the concurrent producer and consumer threads to the code that manipulates the shared data by using **`Monitor`** class methods **`Enter`**, **`Wait`**, **`Pulse`** and **`Exit`**. When a thread uses synchronization to access a shared object, the object is *locked*, so no other thread can acquire the lock for that shared object at the same time.

14.6 Producer/Consumer Relationship with Thread Synchronization

Figure 14.5 demonstrates a producer and a consumer accessing a shared cell of memory with synchronization, so that the consumer consumes only after the producer produces a value and the producer produces a new value only after the consumer consumes the previous value produced. Classes **`Producer`** (lines 90–123), **`Consumer`** (lines 126–162) and **`SharedCell`** (lines 165–200) are identical to Fig. 14.4, except that they use the new class **`HoldIntegerSynchronized`** in this example. [Note: In this example, we demonstrate synchronization with class **`Monitor`**'s **`Enter`** and **`Exit`** methods. In the next example, we demonstrate the same concepts via a **`lock`** block.]

```
1   // Fig. 14.5: Synchronized.cs
2   // Showing multiple threads modifying a shared object with
3   // synchronization.
```

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 1 of 8.)

```

using System;
using System.Threading;

// this class synchronizes access to an integer
public class HoldIntegerSynchronized
{
   // buffer shared by producer and consumer threads
   private int buffer = -1;

   // occupiedBufferCount maintains count of occupied buffers
   private int occupiedBufferCount = 0;

   // property Buffer
   public int Buffer
   {
      get
      {
         // obtain lock on this object
         Monitor.Enter( this );

         // if there is no data to read, place invoking
         // thread in WaitSleepJoin state
         if ( occupiedBufferCount == 0 )
         {
            Console.WriteLine(
               Thread.CurrentThread.Name + " tries to read." );

            DisplayState( "Buffer empty. " +
               Thread.CurrentThread.Name + " waits." );

            Monitor.Wait( this );
         }

         // indicate that producer can store another value
         // because a consumer just retrieved buffer value
         --occupiedBufferCount;

         DisplayState(
            Thread.CurrentThread.Name + " reads " + buffer );

         // tell waiting thread (if there is one) to
         // become ready to execute (Started state)
         Monitor.Pulse( this );

         // Get copy of buffer before releasing lock.
         // It is possible that the producer could be
         // assigned the processor immediately after the
         // monitor is released and before the return
         // statement executes. In this case, the producer
         // would assign a new value to buffer before the
         // return statement returns the value to the
```

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 2 of 8.)

```
         // consumer. Thus, the consumer would receive the
         // new value. Making a copy of buffer and
         // returning the copy ensures that the
         // consumer receives the proper value.
         int bufferCopy = buffer;

         // release lock on this object
         Monitor.Exit( this );

         return bufferCopy;

      } // end get

      set
      {
         // acquire lock for this object
         Monitor.Enter( this );

         // if there are no empty locations, place invoking
         // thread in WaitSleepJoin state
         if ( occupiedBufferCount == 1 )
         {
            Console.WriteLine(
               Thread.CurrentThread.Name + " tries to write." );

            DisplayState( "Buffer full. " +
               Thread.CurrentThread.Name + " waits." );

            Monitor.Wait( this );
         }

         // set new buffer value
         buffer = value;

         // indicate producer cannot store another value
         // until consumer retrieves current buffer value
         ++occupiedBufferCount;

         DisplayState(
            Thread.CurrentThread.Name + " writes " + buffer );

         // tell waiting thread (if there is one) to
         // become ready to execute (Started state)
         Monitor.Pulse( this );

         // release lock on this object
         Monitor.Exit( this );

      } // end set

   }

```

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 3 of 8.)

```
   // display current operation and buffer state
   public void DisplayState( string operation )
   {
      Console.WriteLine( "{0,-35}{1,-9}{2}\n",
         operation, buffer, occupiedBufferCount );
   }

} // end class HoldIntegerSynchronized

// class Producer's Produce method controls a thread that
// stores values from 1 to 4 in sharedLocation
class Producer
{
   private HoldIntegerSynchronized sharedLocation;
   private Random randomSleepTime;

   // constructor
   public Producer(
      HoldIntegerSynchronized shared, Random random )
   {
      sharedLocation = shared;
      randomSleepTime = random;
   }

   // store values 1-4 in object sharedLocation
   public void Produce()
   {
      // sleep for random interval up to 3000 milliseconds
      // then set sharedLocation's Buffer property
      for ( int count = 1; count <= 4; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
         sharedLocation.Buffer = count;
      }

      Console.WriteLine( Thread.CurrentThread.Name +
         " done producing.\nTerminating " +
         Thread.CurrentThread.Name + ".\n" );

   } // end method Produce

} // end class Producer

// class Consumer's Consume method controls a thread that
// loops four times and reads a value from sharedLocation
class Consumer
{
   private HoldIntegerSynchronized sharedLocation;
   private Random randomSleepTime;

```

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 4 of 8.)

```
   // constructor
   public Consumer(
      HoldIntegerSynchronized shared, Random random )
   {
      sharedLocation = shared;
      randomSleepTime = random;
   }

   // read sharedLocation's value four times
   public void Consume()
   {
      int sum = 0;

      // get current thread
      Thread current = Thread.CurrentThread;

      // sleep for random interval up to 3000 milliseconds
      // then add sharedLocation's Buffer property value
      // to sum
      for ( int count = 1; count <= 4; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
         sum += sharedLocation.Buffer;
      }

      Console.WriteLine( Thread.CurrentThread.Name +
         " read values totaling: " + sum +
         ".\nTerminating " + Thread.CurrentThread.Name + ".\n" );

   } // end method Consume

} // end class Consumer

// this class creates producer and consumer threads
class SharedCell
{
   // create producer and consumer threads and start them
   static void Main( string[] args )
   {
      // create shared object used by threads
      HoldIntegerSynchronized holdInteger =
         new HoldIntegerSynchronized();

      // Random object used by each thread
      Random random = new Random();

      // create Producer and Consumer objects
      Producer producer =
         new Producer( holdInteger, random );

      Consumer consumer =
         new Consumer( holdInteger, random );
```

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 5 of 8.)

```

      // output column heads and initial buffer state
      Console.WriteLine( "{0,-35}{1,-9}{2}\n",
         "Operation", "Buffer", "Occupied Count" );
      holdInteger.DisplayState( "Initial state" );

      // create threads for producer and consumer and set
      // delegates for each thread
      Thread producerThread =
         new Thread( new ThreadStart( producer.Produce ) );
      producerThread.Name = "Producer";

      Thread consumerThread =
         new Thread( new ThreadStart( consumer.Consume ) );
      consumerThread.Name = "Consumer";

      // start each thread
      producerThread.Start();
      consumerThread.Start();

   } // end method Main

} // end class SharedCell
```

```
Operation                          Buffer   Occupied C

Initial state                      -1       0

Producer writes 1                  1        1

Consumer reads 1                   1        0

Consumer tries to read.
Buffer empty. Consumer waits.      1        0

Producer writes 2                  2        1

Consumer reads 2                   2        0

Producer writes 3                  3        1

Producer tries to write.
Buffer full. Producer waits.       3        1

Consumer reads 3                   3        0

Producer writes 4                  4        1

Producer done producing.
Terminating Producer.
```

continued on next page

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 6 of 8.)

continued from previous page

```
Consumer reads 4                  4        0

Consumer read values totaling: 10.
Terminating Consumer.
```

```
Operation                         Buffer   Occupied Count

Initial state                     -1       0

Consumer tries to read.
Buffer empty. Consumer waits.     -1       0

Producer writes 1                 1        1

Consumer reads 1                  1        0

Producer writes 2                 2        1

Consumer reads 2                  2        0

Producer writes 3                 3        1

Producer tries to write.
Buffer full. Producer waits.      3        1

Consumer reads 3                  3        0

Producer writes 4                 4        1

Producer done producing.
Terminating Producer.

Consumer reads 4                  4        0

Consumer read values totaling: 10.
Terminating Consumer.
```

```
Operation                         Buffer   Occupied Count

Initial state                     -1       0

Producer writes 1                 1        1

Consumer reads 1                  1        0

Producer writes 2                 2        1
```

continued on next page

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 7 of 8.)

continued from previous page

```
Consumer reads 2                  2        0

Producer writes 3                 3        1

Consumer reads 3                  3        0

Producer writes 4                 4        1

Producer done producing.
Terminating Producer.

Consumer reads 4                  4        0

Consumer read values totaling: 10.
Terminating Consumer.
```

Fig. 14.5 Producer and consumer threads accessing a shared object with synchronization. (Part 8 of 8.)

Class **`HoldIntegerSynchronized`** (lines 9–115) contains two instance variables—**`buffer`** (line 12) and **`occupiedBufferCount`** (line 15). Also, property **`Buffer`**'s **`get`** (lines 20–67) and **`set`** (lines 69–106) accessors now use methods of class **`Monitor`** to synchronize access to property **`Buffer`**. Thus, each object of class **`HoldIntegerSynchronized`** has a *SyncBlock* to maintain synchronization. Instance variable **`occupiedBufferCount`** is known as a *condition variable*—property **`Buffer`**'s accessors use this **`int`** in conditions to determine whether it is the producer's turn to perform a task or the consumer's turn to perform a task. If **`occupiedBufferCount`** is **`0`**, property **`Buffer`**'s **`set`** accessor can place a value into variable **`buffer`**, because the variable currently does not contain information. However, this means that property **`Buffer`**'s **`get`** accessor currently cannot read the value of **`buffer`**. If **`occupiedBufferCount`** is **`1`**, the **`Buffer`** property's **`get`** accessor can read a value from variable **`buffer`**, because the variable currently does contain information. In this case, property **`Buffer`**'s **`set`** accessor currently cannot place a value into **`buffer`**.

As in Fig. 14.4, the producer thread performs the tasks specified in the **`producer`** object's **`Produce`** method. When line 140 sets the value of **`HoldIntegerSynchronized`** property **`Buffer`**, the producer thread invokes the **`set`** accessor at lines 69–104. Line 72 invokes **`Monitor`** method **`Enter`** to acquire the lock on the **`HoldIntegerSynchronized`** object. The **`if`** structure at lines 76–85 determines whether **`occupiedBufferCount`** is **`1`**. If this condition is **`true`**, lines 78–79 output a message indicating that the producer thread tries to write a value, and lines 81–82 invoke method **`DisplayState`** (lines 109–113) to output another message indicating that the buffer is full and that the producer thread waits. Line 84 invokes **`Monitor`** method **`Wait`** to place the calling thread (i.e., the producer) in the *WaitSleepJoin* state for the **`HoldIntegerSynchronized`** object and releases the lock on the object. The *WaitSleepJoin* state for an object is maintained by that object's *SyncBlock*. Now another thread can invoke an accessor method of the **`HoldIntegerSynchronized`** object's **`Buffer`** property.

The producer thread remains in the *WaitSleepJoin* state until the thread is notified that it may proceed—at which point the thread returns to the *Started* state and waits for the system to assign a processor to the thread. When the thread returns to the *Running* state, the thread implicitly reacquires the lock on the `HoldIntegerSynchronized` object and the `set` accessor continues executing with the next statement after `Wait`. Line 88 assigns `value` to `buffer`. Line 92 increments the `occupiedBufferCount` to indicate that the shared buffer now contains a value (i.e., a consumer can read the value, and a producer cannot yet put another value there). Lines 94–95 invoke method `DisplayState` to output a line to the console window indicating that the producer is writing a new value into the `buffer`. Line 99 invokes `Monitor` method `Pulse` with the `HoldIntegerSynchronized` object as an argument. If there are any waiting threads in that object's *SyncBlock*, the first waiting thread enters the *Started* state, indicating that the thread can now attempt its task again (as soon as the thread is assigned a processor). The `Pulse` method returns immediately. Line 102 invokes `Monitor` method `Exit` to release the lock on the `HoldIntegerSynchronized` object, and the `set` accessor returns to its caller.

Common Programming Error 14.4

Forgetting to release the lock on an object when that lock is no longer needed is a logic error. This will prevent the threads in your program that require the lock from acquiring the lock to proceed with their tasks. These threads will be forced to wait (unnecessarily, because the lock is no longer needed). Such waiting can lead to deadlock and indefinite postponement.

The `get` and `set` accessors are implemented similarly. As in Fig. 14.4, the consumer thread performs the tasks specified in the `consumer` object's `Consume` method. The consumer thread gets the value of the `HoldIntegerSynchronized` object's `Buffer` property (line 180) by invoking the `get` accessor at lines 20–67. Line 23 invokes `Monitor` method `Enter` to acquire the lock on the `HoldIntegerSynchronized` object.

The `if` structure at lines 27–36 determines whether `occupiedBufferCount` is `0`. If this condition is `true`, lines 29–30 output a message indicating that the consumer thread tries to read a value, and lines 32–33 invoke method `DisplayState` to output another message indicating that the buffer is empty and that the consumer thread waits. Line 35 invokes `Monitor` method `Wait` to place the calling thread (i.e., the consumer) in the *WaitSleepJoin* state for the `HoldIntegerSynchronized` object and releases the lock on the object. Now another thread can invoke an accessor method of the `HoldIntegerSynchronized` object's `Buffer` property.

The consumer thread object remains in the *WaitSleepJoin* state until the thread is notified that it may proceed—at which point the thread returns to the *Started* state and waits for the system to assign a processor to the thread. When the thread reenters the *Running* state, the thread implicitly reacquires the lock on the `HoldIntegerSynchronized` object, and the `get` accessor continues executing with the next statement after `Wait`. Line 40 decrements `occupiedBufferCount` to indicate that the shared buffer is now empty (i.e., a consumer cannot read the value, but a producer can place another value into the shared buffer), lines 42–43 output a line to the console window indicating the value the consumer is reading and line 47 invokes `Monitor` method `Pulse` with the `HoldIntegerSynchronized` object as an argument. If there are any waiting threads in that object's *SyncBlock*, the first waiting thread enters the *Started* state, indicating that the thread can now attempt its task again (as soon as the thread is assigned a processor). The `Pulse` method returns immediately. Line 60 gets a copy of `buffer` before releasing lock.

It is possible that the producer could be assigned the processor immediately after the lock is released (line 63) and before the **`return`** statement executes (line 65). In this case, the producer would assign a new value to **`buffer`** before the **`return`** statement returns the value to the consumer. Thus, the consumer would receive the new value. Making a copy of **`buffer`** and returning the copy ensures that the consumer receives the proper value. Line 63 invokes **`Monitor`** method **`Exit`** to release the lock on the **`HoldIntegerSynchronized`** object and the **`get`** accessor returns **`bufferCopy`** to its caller.

Study the outputs in Fig. 14.5. Observe that every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once. This occurs because the producer and consumer cannot perform tasks unless it is “their turn.” The producer must go first; the consumer must wait if the producer has not produced, since the consumer last consumed; and the producer must wait if the consumer has not yet consumed the value the producer most recently produced. Execute this program several times to confirm that every integer produced is consumed once.

In the first and second sample outputs, notice the lines indicating when the producer and consumer must wait to perform their respective tasks. In the third sample output, notice that the producer and consumer were able to perform their tasks without waiting.

14.7 Producer/Consumer Relationship: Circular Buffer

Figure 14.5 uses thread synchronization to guarantee that two threads manipulate data in a shared buffer correctly. However, the application may not perform optimally. If the two threads operate at different speeds, one of the threads will spend more (or most) of its time waiting. For example, in Fig. 14.5 we shared a single integer between the two threads. If the producer thread produces values faster than the consumer can consume those values, then the producer thread waits for the consumer, because there are no other locations in memory to place the next value. Similarly, if the consumer consumes faster than the producer can produce values, the consumer waits until the producer places the next value into the shared location in memory. Even when we have threads that operate at the same relative speeds, over a period of time, those threads may become “out of sync,” causing one of the threads to wait for the other. We cannot make assumptions about the relative speeds of asynchronous concurrent threads. There are too many interactions that occur with the operating system, the network, the user and other components, which can cause the threads to operate a different speeds. When this happens, threads wait. When threads wait, programs become less productive, user-interactive programs become less responsive and network applications suffer longer delays because the processor is not used efficiently.

To minimize the waiting for threads that share resources and operate at the same relative speeds, we can implement a *circular buffer* that provides extra buffers into which the producer can place values and from which the consumer can retrieve those values. Let us assume the buffer is implemented as an array. The producer and consumer work from the beginning of the array. When either thread reaches the end of the array, it simply returns to the first element of the array to perform its next task. If the producer temporarily produces values faster than the consumer can consume them, the producer can write additional values into the extra buffers (if cells are available). This enables the producer to perform its task even though the consumer is not ready to receive the current value being produced. Similarly, if the consumer consumes faster than the producer produces new values, the consumer can read additional values from the buffer (if there are any). This enables the consumer to perform its task even though the producer is not ready to produce additional values.

Note that the circular buffer would be inappropriate if the producer and consumer operate at different speeds. If the consumer always executes faster than the producer, then a buffer at one location is enough. Additional locations would waste memory. If the producer always executes faster, a buffer with an infinite number of locations would be requires to absorb the extra production.

The key to using a circular buffer is to define it with enough extra cells to handle the anticipated “extra” production. If, over a period of time, we determine that the producer often produces as many as three more values than the consumer can consume, we can define a buffer of at least three cells to handle the extra production. We do not want the buffer to be too small, because that would cause threads to wait more. On the other hand, we do not want the buffer to be too large, because that would waste memory.

Performance Tip 14.4

Even when using a circular buffer, it is possible that a producer thread could fill the buffer, which would force the producer thread to wait until a consumer consumes a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, the consumer thread must wait until the producer produces another value. The key to using a circular buffer is optimizing the buffer size to minimize the amount of thread-wait time.

Figure 14.6 demonstrates a producer and a consumer accessing a circular buffer (in this case, a shared array of two cells) with synchronization. In this version of the producer/consumer relationship, the consumer consumes a value only when the array is not empty and the producer produces a value only when the array is not full. This program is implemented as a Windows application that sends its output to a **`TextBox`**. Classes **`Producer`** (lines 174–210) and **`Consumer`** (lines 213–252) perform the same tasks as in Fig. 14.4 and Fig. 14.5, except that they output messages to the **`TextBox`** in the application window. The statements that created and started the thread objects in the **`Main`** methods of class **`SharedCell`** in Fig. 14.4 and Fig. 14.5 now appear in class **`CircularBuffer`** (lines 255–313), where the **`Load`** event handler (lines 278–311) performs the statements.

The most significant changes from Fig. 14.5 occur in class **`HoldIntegerSynchronized`**, which now contains six instance variables. Array **`buffers`** is a three-element integer array that represents the circular buffer. Variable **`occupiedBufferCount`** is the condition variable that can be used to determine whether a producer can write into the circular buffer (i.e., **`occupiedBufferCount`** is less than the number of elements in array **`buffers`**) and whether a consumer can read from the circular buffer (i.e., **`occupiedBufferCount`** is greater than **`0`**). Variable **`readLocation`** indicates the position from which the next value can be read by a consumer. Variable **`writeLocation`** indicates the next location in which a value can be placed by a producer. The program displays output in **`outputTextBox`** (a **`TextBox`** control).

```
1  // Fig. 14.6: CircularBuffer.cs
2  // Implementing the producer/consumer relationship with a
3  // circular buffer.
```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 1 of 9.)

```

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Threading;

// implement the shared integer with synchronization
public class HoldIntegerSynchronized
{
   // each array element is a buffer
   private int[] buffers = { -1, -1, -1 };

   // occupiedBufferCount maintains count of occupied buffers
   private int occupiedBufferCount = 0;

   // variable that maintain read and write buffer locations
   private int readLocation = 0, writeLocation = 0;

   // GUI component to display output
   private TextBox outputTextBox;

   // constructor
   public HoldIntegerSynchronized( TextBox output )
   {
      outputTextBox = output;
   }

   // property Buffer
   public int Buffer
   {
      get
      {
         // lock this object while getting value
         // from buffers array
         lock ( this )
         {
            // if there is no data to read, place invoking
            // thread in WaitSleepJoin state
            if ( occupiedBufferCount == 0 )
            {
               outputTextBox.Text += "\r\nAll buffers empty. " +
                  Thread.CurrentThread.Name + " waits.";
               outputTextBox.ScrollToCaret();

               Monitor.Wait( this );
            }

            // obtain value at current readLocation, then
            // add string indicating consumed value to output
            int readValue = buffers[ readLocation ];
```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 2 of 9.)

```

               outputTextBox.Text += "\r\n" +
                  Thread.CurrentThread.Name + " reads " +
                  buffers[ readLocation ] + " ";

               // just consumed a value, so decrement number of
               // occupied buffers
               --occupiedBufferCount;

               // update readLocation for future read operation,
               // then add current state to output
               readLocation =
                  ( readLocation + 1 ) % buffers.Length;
               outputTextBox.Text += CreateStateOutput();
               outputTextBox.ScrollToCaret();

               // return waiting thread (if there is one)
               // to Started state
               Monitor.Pulse( this );

               return readValue;

            } // end lock

         } // end accessor get

         set
         {
            // lock this object while setting value
            // in buffers array
            lock ( this )
            {
               // if there are no empty locations, place invoking
               // thread in WaitSleepJoin state
               if ( occupiedBufferCount == buffers.Length )
               {
                  outputTextBox.Text += "\r\nAll buffers full. " +
                     Thread.CurrentThread.Name + " waits.";
                  outputTextBox.ScrollToCaret();

                  Monitor.Wait( this );
               }

               // place value in writeLocation of buffers, then
               // add string indicating produced value to output
               buffers[ writeLocation ] = value;

               outputTextBox.Text += "\r\n" +
                  Thread.CurrentThread.Name + " writes " +
                  buffers[ writeLocation ] + " ";

```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 3 of 9.)

```
            // just produced a value, so increment number of
            // occupied buffers
            ++occupiedBufferCount;

            // update writeLocation for future write operation,
            // then add current state to output
            writeLocation =
               ( writeLocation + 1 ) % buffers.Length;
            outputTextBox.Text += CreateStateOutput();
            outputTextBox.ScrollToCaret();

            // return waiting thread (if there is one)
            // to Started state
            Monitor.Pulse( this );

         } // end lock

      } // end accessor set

   } // end property Buffer

   // create state output
   public string CreateStateOutput()
   {
      // display first line of state information
      string output = "(buffers occupied: " +
         occupiedBufferCount + ")\r\nbuffers: ";

      for ( int i = 0; i < buffers.Length; i++ )
         output += " " + buffers[ i ] + "  ";

      output += "\r\n";

      // display second line of state information
      output += "         ";

      for ( int i = 0; i < buffers.Length; i++ )
         output += "---- ";

      output += "\r\n";

      // display third line of state information
      output += "         ";

      // display readLocation (R) and writeLocation (W)
      // indicators below appropriate buffer locations
      for ( int i = 0; i < buffers.Length; i++ )

         if ( i == writeLocation &&
               writeLocation == readLocation )
            output += " WR  ";
         else if ( i == writeLocation )
            output += " W   ";
```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 4 of 9.)

```
         else if  ( i == readLocation )
            output += "  R  ";
         else
            output += "     ";

      output += "\r\n";

      return output;
   }

} // end class HoldIntegerSynchronized

// produce the integers from 11 to 20 and place them in buffer
public class Producer
{
   private HoldIntegerSynchronized sharedLocation;
   private TextBox outputTextBox;
   private Random randomSleepTime;

   // constructor
   public Producer( HoldIntegerSynchronized shared,
      Random random, TextBox output )
   {
      sharedLocation = shared;
      outputTextBox = output;
      randomSleepTime = random;
   }

   // produce values from 11-20 and place them in
   // sharedLocation's buffer
   public void Produce()
   {
      // sleep for random interval up to 3000 milliseconds
      // then set sharedLocation's Buffer property
      for ( int count = 11; count <= 20; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
         sharedLocation.Buffer = count;
      }

      string name = Thread.CurrentThread.Name;

      outputTextBox.Text += "\r\n" + name +
         " done producing.\r\n" + name + " terminated.\r\n";

      outputTextBox.ScrollToCaret();

   } // end method Produce

} // end class Producer

```

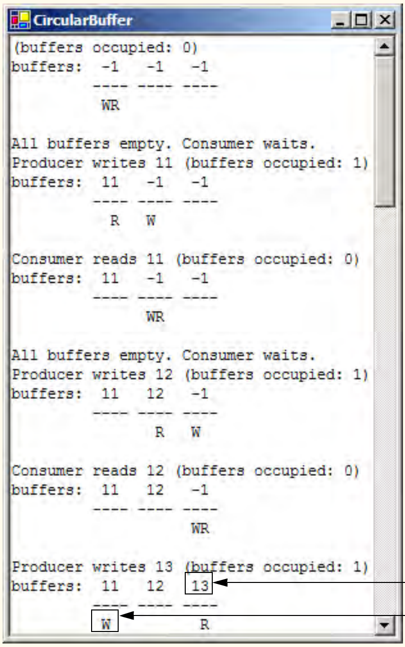
Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 5 of 9.)

```
// consume the integers 1 to 10 from circular buffer
public class Consumer
{
   private HoldIntegerSynchronized sharedLocation;
   private TextBox outputTextBox;
   private Random randomSleepTime;

   // constructor
   public Consumer( HoldIntegerSynchronized shared,
      Random random, TextBox output )
   {
      sharedLocation = shared;
      outputTextBox = output;
      randomSleepTime = random;
   }

   // consume 10 integers from buffer
   public void Consume()
   {
      int sum = 0;

      // loop 10 times and sleep for random interval up to
      // 3000 milliseconds then add sharedLocation's
      // Buffer property value to sum
      for ( int count = 1; count <= 10; count++ )
      {
         Thread.Sleep( randomSleepTime.Next( 1, 3000 ) );
         sum += sharedLocation.Buffer;
      }

      string name = Thread.CurrentThread.Name;

      outputTextBox.Text += "\r\nTotal " + name +
         " consumed: " + sum + ".\r\n" + name +
         " terminated.\r\n";

      outputTextBox.ScrollToCaret();

   } // end method Consume

} // end class Consumer

// set up the producer and consumer and start them
public class CircularBuffer : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox outputTextBox;

   // required designer variable
   private System.ComponentModel.Container components = null;

```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 6 of 9.)

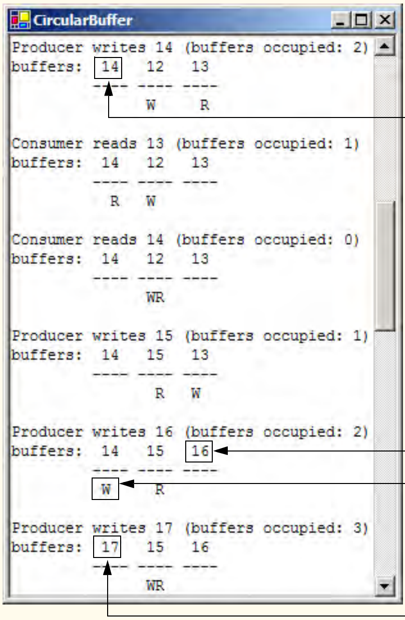
```
      // no-argument constructor
      public CircularBuffer()
      {
         InitializeComponent();
      }

      // Visual Studio .NET GUI code appears here in source file

      // main entry point for the application
      [STAThread]
      static void Main()
      {
         Application.Run( new CircularBuffer() );
      }

      // Load event handler creates and starts threads
      private void CircularBuffer_Load(
         object sender, System.EventArgs e )
      {
         // create shared object
         HoldIntegerSynchronized sharedLocation =
            new HoldIntegerSynchronized( outputTextBox );

         // display sharedLocation state before producer
         // and consumer threads begin execution
         outputTextBox.Text = sharedLocation.CreateStateOutput();

         // Random object used by each thread
         Random random = new Random();

         // create Producer and Consumer objects
         Producer producer =
            new Producer( sharedLocation, random, outputTextBox );
         Consumer consumer =
            new Consumer( sharedLocation, random, outputTextBox );

         // create and name threads
         Thread producerThread =
            new Thread( new ThreadStart( producer.Produce ) );
         producerThread.Name = "Producer";

         Thread consumerThread =
            new Thread( new ThreadStart( consumer.Consume ) );
         consumerThread.Name = "Consumer";

         // start threads
         producerThread.Start();
         consumerThread.Start();

      } // end CircularBuffer_Load method

   } // end class CircularBuffer
```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 7 of 9.)



```
(buffers occupied: 0)
buffers:  -1   -1   -1
         ---- ---- ----
          WR

All buffers empty. Consumer waits.
Producer writes 11 (buffers occupied: 1)
buffers:  11   -1   -1
         ---- ---- ----
           R   W

Consumer reads 11 (buffers occupied: 0)
buffers:  11   -1   -1
         ---- ---- ----
               WR

All buffers empty. Consumer waits.
Producer writes 12 (buffers occupied: 1)
buffers:  11   12   -1
         ---- ---- ----
                R   W

Consumer reads 12 (buffers occupied: 0)
buffers:  11   12   -1
         ---- ---- ----
                    WR

Producer writes 13 (buffers occupied: 1)
buffers:  11   12   13
         ---- ---- ----
          W          R
```

Value placed in last buffer.
Next value will be deposited in leftmost buffer.



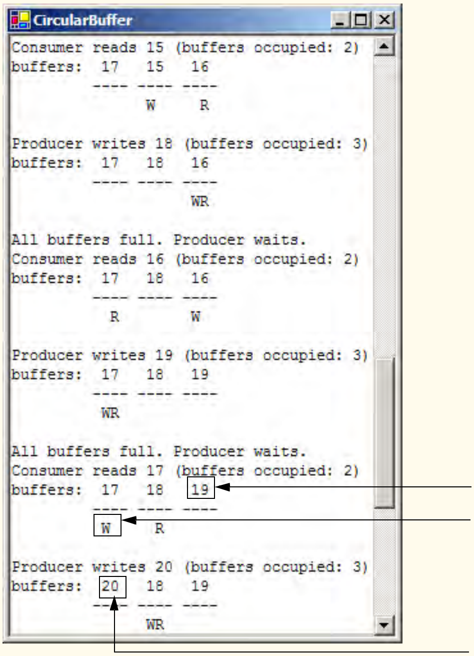
```
Producer writes 14 (buffers occupied: 2)
buffers:  14   12   13
         ---- ---- ----
               W     R

Consumer reads 13 (buffers occupied: 1)
buffers:  14   12   13
         ---- ---- ----
           R   W

Consumer reads 14 (buffers occupied: 0)
buffers:  14   12   13
         ---- ---- ----
               WR

Producer writes 15 (buffers occupied: 1)
buffers:  14   15   13
         ---- ---- ----
                R   W

Producer writes 16 (buffers occupied: 2)
buffers:  14   15   16
         ---- ---- ----
          W     R

Producer writes 17 (buffers occupied: 3)
buffers:  17   15   16
         ---- ---- ----
               WR
```

Circular buffer effect—the fourth value is deposited in the leftmost buffer.

Value placed in last buffer.
Next value will be deposited in leftmost buffer.

Circular buffer effect—the seventh value is deposited in the leftmost buffer.

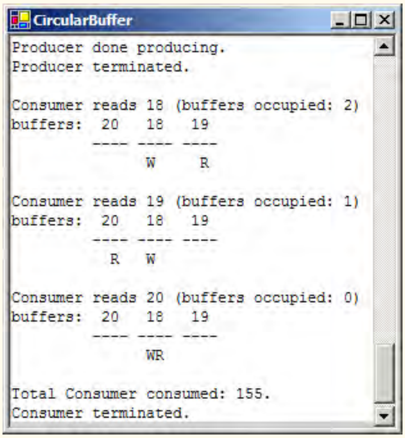
Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 8 of 9.)



```
Consumer reads 15 (buffers occupied: 2)
buffers:  17   15   16
         ---- ---- ----
               W     R

Producer writes 18 (buffers occupied: 3)
buffers:  17   18   16
         ---- ---- ----
                    WR

All buffers full. Producer waits.
Consumer reads 16 (buffers occupied: 2)
buffers:  17   18   16
         ---- ---- ----
           R        W

Producer writes 19 (buffers occupied: 3)
buffers:  17   18   19
         ---- ---- ----
          WR

All buffers full. Producer waits.
Consumer reads 17 (buffers occupied: 2)
buffers:  17   18   19
         ---- ---- ----
          W     R

Producer writes 20 (buffers occupied: 3)
buffers:  20   18   19
         ---- ---- ----
               WR
```

Value placed in last buffer. Next value will be deposited in leftmost buffer.

Circular buffer effect—the tenth value is deposited in the leftmost buffer.



```
Producer done producing.
Producer terminated.

Consumer reads 18 (buffers occupied: 2)
buffers:  20   18   19
         ---- ---- ----
               W     R

Consumer reads 19 (buffers occupied: 1)
buffers:  20   18   19
         ---- ---- ----
           R   W

Consumer reads 20 (buffers occupied: 0)
buffers:  20   18   19
         ---- ---- ----
               WR

Total Consumer consumed: 155.
Consumer terminated.
```

Fig. 14.6 Producer and consumer threads accessing a circular buffer. (Part 9 of 9.)

The `set` accessor (lines 83–125) of property `Buffer` performs the same tasks that it did in Fig. 14.5, with a few modifications. Rather than using `Monitor` methods `Enter` and `Exit` to acquire and release the lock on the `HoldIntegerSynchronized` object, we use a block of code preceded by keyword `lock` to lock the `HoldIntegerSynchronized` object. As program control enters the `lock` block, the currently executing thread acquires the lock (assuming the lock is currently available) on the `HoldIntegerSynchronized` object (i.e., `this`). When the `lock` block terminates, the thread releases the lock automatically.

Common Programming Error 14.5

When using class `Monitor`*'s* `Enter` *and* `Exit` *methods to manage an object's lock,* `Exit` *must be called explicitly to release the lock. If an exception occurs in a method before* `Exit` *can be called and that exception is not caught, the method could terminate without calling* `Exit`*. If so, the lock is not released. To avoid this error, place code that could throw exceptions in a* `try` *block, and place the call to* `Exit` *in the corresponding* `finally` *block to ensure that the lock is released.*

Software Engineering Observation 14.2

Using a `lock` *block to manage the lock on a synchronized object eliminates the possibility of forgetting to relinquish the lock with a call to* `Monitor` *method* `Exit`*. C# implicitly calls* `Monitor` *method* `Exit` *when a* `lock` *block terminates for any reason. Thus, even if an exception occurs in the block, the lock will be released.*

The `if` structure at lines 91–98 in the `set` accessor determines whether the producer must wait (i.e., all buffers are full). If the producer thread must wait, lines 93–94 append text to the `outputTextBox` indicating that the producer is waiting to perform its task and line 97 invokes `Monitor` method `Wait` to place the producer thread in the *WaitSleepJoin* state of the `HoldIntegerSynchronized` object. When execution continues at line 102 after the `if` structure, the value written by the producer is placed in the circular buffer at location `writeLocation`. Next, lines 104–106 append a message containing the value produced to the `TextBox`. Line 110 increments `occupiedBufferCount`, because there is now at least one value in the buffer that the consumer can read. Then, lines 114–115 update `writeLocation` for the next call to the `set` accessor of property `Buffer`. The output continues at line 116 by invoking method `CreateStateOutput` (lines 130–169), which outputs the number of occupied buffers, the contents of the buffers and the current `writeLocation` and `readLocation`. Finally, line 121 invokes `Monitor` method `Pulse` to indicate that a thread waiting on the `HoldIntegerSynchronized` object (if there is a waiting thread) should transition to the *Started* state. Note that reaching the closing right brace of the `lock` block at line 123 causes the thread to release the lock on the `HoldIntegerSynchronized` object.

The `get` accessor (lines 37–81) of property `Buffer` also performs the same tasks in this example that it did in Fig. 14.5, with a few minor modifications. The `if` structure at lines 45–52 in the `get` accessor determines whether the consumer must wait (i.e., all buffers are empty). If the consumer thread must wait, lines 47–48 append text to the `outputTextBox` indicating that the consumer is waiting to perform its task, and line 51 invokes `Monitor` method `Wait` to place the consumer thread in the *WaitSleepJoin* state of the `HoldIntegerSynchronized` object. Once again, we use a `lock` block to acquire and release the lock on the `HoldIntegerSynchronized` object, rather than using `Monitor` methods `Enter` and `Exit`. When execution continues at line 56 after the `if` structure, `readValue` is assigned the value at location `readLocation` in the circular buffer. Lines 58–60 appends the value consumed to the `TextBox`. Line 64 decrements the `occupiedBufferCount`, because there is at least one open position in the buffer in which the producer thread can place a value. Then, lines 68–69 update `readLocation` for the next call to the `get` accessor of `Buffer`. Line 70 invokes method `CreateStateOutput` to output the number of occupied buffers, the contents of the buffers and the current `writeLocation` and `readLocation`. Finally, line 80 invokes method `Pulse` to transition the next thread waiting for the `HoldIntegerSynchro-`

`nized` object into the *Started* state, and line 82 returns the consumed value to the calling method.

In Fig. 14.6, the outputs include the current **`occupiedBufferCount`**, the contents of the buffers and the current **`writeLocation`** and **`readLocation`**. In the output, the letters **`W`** and **`R`** represent the current **`writeLocation`** and **`readLocation`**, respectively. Notice that, after the third value is placed in the third element of the buffer, the fourth value is inserted at the beginning of the array. This provides the circular buffer effect.

SUMMARY

- Computers perform operations concurrently, such as compiling programs, printing files and receiving electronic mail messages over a network.
- Programming languages generally provide only a simple set of control structures that enable programmers to perform one action at a time and proceed to the next action after the previous one finishes.
- Historically, the concurrency that computers perform generally has been implemented as operating system "primitives" available only to highly experienced "systems programmers."
- The .NET Framework Class Library makes concurrency primitives available to the applications programmer. The programmer specifies that applications contain threads of execution, each thread designating a portion of a program that may execute concurrently with other threads—this capability is called multithreading.
- A thread that was just created is in the *Unstarted* state. A thread is initialized using the **`Thread`** class's constructor which receives a **`ThreadStart`** delegate. This delegate specifies the method that contains the tasks a thread will perform.
- A thread remains in the *Unstarted* state until the thread's **`Start`** method is called; this causes the thread to enter the *Started* state (also known as the *Ready* or *Runnable* state).
- A thread in the *Started* state enters the *Running* state when the system assigns a processor to the thread. The system assigns the processor to the highest-priority *Started* thread.
- A thread enters the *Stopped* (or *Dead*) state when its **`ThreadStart`** delegate completes or terminates. A thread is forced into the *Stopped* state when its **`Abort`** method is called (by itself or by another thread).
- A *Running* thread enters the *Blocked* state when the thread issues an input/output request. A *Blocked* thread becomes *Started* when the I/O it is waiting for completes. A *Blocked* thread cannot use a processor, even if one is available.
- If a thread wants to go to sleep, it calls **`Thread`** method **`Sleep`**. A thread wakes up when the designated sleep interval expires.
- If a thread cannot continue executing (we will call this the dependent thread) unless another thread terminates, the dependent thread calls the other thread's ***`Join`*** method to "join" the two threads. When two threads are "joined," the dependent thread leaves the *WaitSleepJoin* state when the other thread finishes execution (enters the *Stopped* state).
- In thread synchronization, when a thread encounters code that it cannot yet run (e.g., a producer cannot produce at the current time), the thread can call **`Monitor`** method **`Wait`** until certain actions occur that enable the thread to continue executing.
- Any thread in the *WaitSleepJoin* state can leave that state if another thread invokes **`Thread`** method **`Interrupt`** on the thread in the *WaitSleepJoin* state.
- If a thread called **`Monitor`** method **`Wait`**, a corresponding call to the **`Monitor`** method **`Pulse`** or **`PulseAll`** by another thread in the program will transition the original thread from the *WaitSleepJoin* state to the *Started* state.

- If **Thread** method **Suspend** is called on a thread (by the thread itself or by another thread in the program), the thread enters the *Suspended* state. A thread leaves the *Suspended* state when a separate thread invokes **Thread** method **Resume** on the suspended thread.
- Every C# thread has a priority of **ThreadPriority.Lowest**, **ThreadPriority.BelowNormal**, **ThreadPriority.Normal**, **ThreadPriority.AboveNormal** or **ThreadPriority.Highest**.
- The job of the thread scheduler is to keep the highest-priority thread running at all times and, if there is more than one highest-priority thread, to ensure that all equally high-priority threads execute for a quantum at a time in round-robin fashion.
- A thread's priority can be adjusted with the **Priority** property, which accepts an argument from the **ThreadPriority** enumeration.
- A thread that updates shared data calls **Monitor** method **Enter** to acquire the lock on that data. It updates the data and calls **Monitor** method **Exit** upon completion of the update. While that data is locked, all other threads attempting to acquire the lock on that data must wait.
- If you place the **lock** keyword before a block of code the lock is acquired on the specified object as program control enters the block; the lock is released when the block terminates for any reason.
- If a thread decides that it cannot continue execution, it can call **Wait**. This puts the thread into the *WaitSleepJoin* state. When the thread can continue execution again, **Pulse** or **PulseAll** is called to notify the thread to continue running.
- When the **lock** keyword is used, C# implicitly calls the **Exit** method whenever we leave the scope of the block.

TERMINOLOGY

Abort method of class **Thread**
AboveNormal constant in **ThreadPriority**
accessing shared data with synchronization
acquire the lock for an object
automatic garbage collection
BelowNormal constant in **ThreadPriority**
Blocked state
Blocked thread
built-in multithreading
circular buffer
concurrency
concurrent producer and consumer threads
concurrent programming
condition variable
consumer
Dead state
deadlock
Enter method of class **Monitor**
Exit method of class **Monitor**
garbage collection
garbage-collector thread
Highest constant in **ThreadPriority**
Hoare, C. A. R.
I/O completion
I/O request
indefinite postponement
input/output blocking
Interrupt method of class **Thread**
Join method of class **Thread**
life cycle of a thread
lock keyword
locking objects
Lowest constant in **ThreadPriority**
memory leak
Monitor class
multilevel priority queue
multithreading
Name property of class **Thread**
Normal constant in **ThreadPriority**
Priority property of class **Thread**
priority scheduling
producer
producer/consumer relationship
Pulse method of class **Monitor**
PulseAll method of class **Monitor**
quantum
quantum expiration
Ready state
release a lock
Resume method of class **Thread**

Runnable state
Running state
scheduling
shared buffer
sleep interval expires
`Sleep` method of class `Thread`
sleeping thread
`Start` method of class `Thread`
Started state
starvation
Stopped state
`Suspend` method of class `Thread`
SyncBlock
synchronized block of code
`System.Threading` namespace
task
`Thread` class
thread of execution
thread-priority scheduling
thread state
`ThreadAbortException`
`ThreadPriority` enumeration
`ThreadStart` delegate
Unstarted state
`Wait` method of class `Monitor`
WaitSleepJoin state

SELF-REVIEW EXERCISES

14.1 Fill in the blanks in each of the following statements:

a) Monitor methods __________ and __________ acquire and release the lock on an object.
b) Among a group of equal-priority threads, each thread receives a brief burst of time called a __________, during which the thread has the processor and can perform its tasks.
c) C# provides a __________ thread that reclaims dynamically allocated memory.
d) Four reasons a thread that is alive is not in the *Started* state are __________, __________, __________ and __________.
e) A thread enters the __________ state when the method that controls the thread’s lifecycle terminates.
f) A thread’s priority must be one of the `ThreadPriority` constants __________, __________, __________, __________ and __________.
g) To wait for a designated number of milliseconds then resume execution, a thread should call the __________ method of class `Thread`.
h) Method __________ of class `Monitor` transitions a thread in the *WaitSleepJoin* state to the *Started* state.
i) A __________ block automatically acquires the lock on an object as the program control enters the block and releases the lock on that object when the block terminates execution.
j) Class `Monitor` provides methods that __________ access to shared data.

14.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) A thread cannot execute if it is in the *Stopped* state.
b) In C#, a higher priority thread entering (or reentering) the *Started* state will preempt threads of lower priority.
c) The code that a thread executes is defined in its `Main` method.
d) A thread in the *WaitSleepJoin* state always returns to the *Started* state when `Monitor` method `Pulse` is called.
e) Method `Sleep` of class `Thread` does not consume processor time while a thread sleeps.
f) A blocked thread can be placed in the *Started* state by `Monitor` method `Pulse`.
g) Class `Monitor`’s `Wait`, `Pulse` and `PulseAll` methods can be used in any block of code.
h) The programmer must place a call to `Monitor` method `Exit` in a `lock` block to relinquish the lock.
i) When `Monitor` class method `Wait` is called within a `lock`ed block, the lock for that block is released and the thread that called `Wait` is placed in the *WaitSleepJoin* state.

ANSWERS TO SELF-REVIEW EXERCISES

14.1 a) **`Enter`**, **`Exit`**. b) timeslice or quantum. c) garbage collector. d) waiting, sleeping, suspended, blocked for input/output. e) *Stopped*. f) **`Lowest`**, **`BelowNormal`**, **`Normal`**, **`AboveNormal`**, **`Highest`**. g) **`Sleep`**. h) **`Pulse`**. i) **`lock`**. j) synchronize.

14.2 a) True. b) True. c) False. The code that a thread executes is defined in the method specified by the thread's **`ThreadStart`** delegate. d) False. A thread may be in the *WaitSleepJoin* state for several reasons. Calling Pulse moves a thread from the *WaitSleepJoin* state to the *Started* state only if the thread entered the *WaitSleepJoin* state as the result of a call to **`Monitor`** method **`Pulse`**. e) True. f) False. A thread is blocked by the operating system and returns to the *Started* state when the operating system determines that the thread can continue executing (e.g., when an I/O request completes or when a lock the thread attempted to acquire becomes available). g) False. Class **`Monitor`** methods can be called only if the thread performing the call currently owns the lock on the object each method receives as an argument. h) False. A **`lock`** block implicitly relinquishes the lock when the thread completes execution of the **`lock`** block. i) True.

EXERCISES

14.3 The code that manipulates the circular buffer in Fig. 14.6 will work with a buffer of two or more elements. Try changing the buffer size to see how it affects the producer and consumer threads. In particular, notice that the producer waits to produce less frequently as the buffer grows in size.

14.4 Write a program to demonstrate that, as a high-priority thread executes, it will delay the execution of all lower-priority threads.

14.5 Write a program that demonstrates timeslicing among several equal-priority threads. Show that a lower-priority thread's execution is deferred by the timeslicing of the higher-priority threads.

14.6 Write a program that demonstrates a high-priority thread using **`Sleep`** to give lower-priority threads a chance to run.

14.7 Two problems that can occur in languages like C# that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time but may eventually complete. Give an example of how each of these problems can occur in a multithreaded C# program.

14.8 (Readers and Writers) This exercise asks you to develop a C# monitor to solve a famous problem in concurrency control. This problem was first discussed and solved by P. J. Courtois, F. Heymans and D. L. Parnas in their research paper, "Concurrent Control with Readers and Writers," *Communications of the ACM*, Vol. 14, No. 10, October 1971, pp. 667–668. The interested student might also want to read C. A. R. Hoare's seminal research paper on monitors, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17, No. 10, October 1974, pp. 549–557. *Corrigendum, Communications of the ACM*, Vol. 18, No. 2, February 1975, p. 95. [The readers and writers problem is discussed at length in Chapter 5 of the author's book: Deitel, H. M., *Operating Systems*, Reading, MA: Addison-Wesley, 1990.]

With multithreading, many threads can access shared data; as we have seen, access to shared data needs to be synchronized to avoid corrupting the data.

Consider an airline-reservation system in which many clients are attempting to book seats on particular flights between particular cities. All the information about flights and seats is stored in a common database in memory. The database consists of many entries, each representing a seat on a particular flight for a particular day between particular cities. In a typical airline-reservation scenario, the client will probe the database looking for the "optimal" flight to meet that client's needs. A client may probe the database many times before trying to book a particular flight. A seat that was available during this probing phase could easily be booked by someone else before the client has a chance to book it after deciding on it. In that case, when the client attempts to make the reservation, the client will discover that the data has changed and the flight is no longer available.

The client probing the database is called a *reader*. The client attempting to book the flight is called a *writer*. Any number of readers can be probing shared data at once, but each writer needs exclusive access to the shared data to prevent the data from being corrupted.

Write a multithreaded C# program that launches multiple reader threads and multiple writer threads, each attempting to access a single reservation record. A writer thread has two possible transactions, **`makeReservation`** and **`cancelReservation`**. A reader has one possible transaction, **`queryReservation`**.

First, implement a version of your program that allows unsynchronized access to the reservation record. Show how the integrity of the database can be corrupted. Next, implement a version of your program that uses C# monitor synchronization with **`Wait`** and **`Pulse`** to enforce a disciplined protocol for readers and writers accessing the shared reservation data. In particular, your program should allow multiple readers to access the shared data simultaneously when no writer is active—but, if a writer is active, then no reader should be allowed to access the shared data.

Be careful. This problem has many subtleties. For example, what happens when there are several active readers and a writer wants to write? If we allow a steady stream of readers to arrive and share the data, they could indefinitely postpone the writer (who might become tired of waiting and take his or her business elsewhere). To solve this problem, you might decide to favor writers over readers. But here, too, there is a trap, because a steady stream of writers could then indefinitely postpone the waiting readers, and they, too, might choose to take their business elsewhere! Implement your monitor with the following methods: **`startReading`**, which is called by any reader who wants to begin accessing a reservation; **`stopReading`**, to be called by any reader who has finished reading a reservation; **`startWriting`**, to be called by any writer who wants to make a reservation; and **`stopWriting`**, to be called by any writer who has finished making a reservation.

15

Strings, Characters and Regular Expressions

Objectives

- To be able to create and manipulate immutable character string objects of class **`String`**.
- To be able to create and manipulate mutable character string objects of class **`StringBuilder`**.
- To be able to use regular expressions in conjunction with classes **`Regex`** and **`Match`**.

The chief defect of Henry King
Was chewing little bits of string.
Hilaire Belloc

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences.
William Strunk, Jr.

I have made this letter longer than usual, because I lack the time to make it short.
Blaise Pascal

The difference between the almost-right word & the right word is really a large matter—it's the difference between the lightning bug and the lightning.
Mark Twain

Mum's the word.
Miguel de Cervantes, *Don Quixote de la Mancha*

Outline

15.1 Introduction
15.2 Fundamentals of Characters and Strings
15.3 `String` Constructors
15.4 `String Indexer`, `Length` Property and `CopyTo` Method
15.5 Comparing `Strings`
15.6 `String` Method `GetHashCode`
15.7 Locating Characters and Substrings in `Strings`
15.8 Extracting Substrings from `Strings`
15.9 Concatenating `Strings`
15.10 Miscellaneous `String` Methods
15.11 Class `StringBuilder`
15.12 `StringBuilder Indexer`, `Length` and `Capacity` Properties, and `EnsureCapacity` Method
15.13 `StringBuilder Append` and `AppendFormat` Methods
15.14 `StringBuilder Insert`, `Remove` and `Replace` Methods
15.15 `Char` Methods
15.16 Card Shuffling and Dealing Simulation
15.17 Regular Expressions and Class `Regex`

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

15.1 Introduction

In this chapter, we introduce the Framework Class Library's string and character processing capabilities and demonstrate the use of regular expressions to search for patterns in text. The techniques presented in this chapter can be employed to develop text editors, word processors, page-layout software, computerized typesetting systems and other kinds of text-processing software. Previous chapters have already presented several string-processing capabilities. In this chapter, we expand on this information by detailing the capabilities of class ***`String`*** and type ***`char`*** from the **`System`** namespace, class ***`StringBuilder`*** from the ***`System.Text`*** namespace and classes ***`Regex`*** and ***`Match`*** from the ***`System.Text.RegularExpressions`*** namespace.

15.2 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of C# source code. Every program is composed of characters that, when grouped together meaningfully, create a sequence that the compiler interprets as a series of instructions that describe how to accomplish a task. In addition to normal characters, a program also can contain *character constants*. A character constant is a character that is represented as an integer value, called a *character code*. For example, the integer value `122` corresponds to the character constant `'z'`. The integer value `10` corresponds to the new line character `'\n'`. Character constants are established according to the *Unicode character set*, an international character set that contains many more symbols and letters than does the ASCII character set (see Appendix F, ASCII character set). To learn more about Unicode, see Appendix G, Unicode.

A string is a series of characters treated as a single unit. These characters can be uppercase letters, lowercase letters, digits and various *special characters*, such as `+`, `-`, `*`, `/`, `$` and others. A string is an object of class `String` in the `System` namespace.[1] We write *string literals*, or *string constants* (often called *literal* ***`string`****s*), as sequences of characters in double quotation marks, as follows:

```
"John Q. Doe"
"9999 Main Street"
"Waltham, Massachusetts"
"(201) 555-1212"
```

A declaration can assign a `string` literal to a `string` reference. The declaration

```
string color = "blue";
```

initializes `string` reference `color` to refer to the `string` literal object `"blue"`.

Performance Tip 15.1

If there are multiple occurrences of the same ***`string`*** *literal object in an application, a single copy of the* ***`string`*** *literal object will be referenced from each location in the program that uses that* ***`string`*** *literal. It is possible to share the object in this manner, because* ***`string`*** *literal objects are implicitly constant. Such sharing conserves memory.*

On occasion, a `string` will contain multiple backslash characters (this often occurs in the name of a file). It is possible to exclude escape sequences and interpret all the characters in a `string` literally, using the `@` character. Backslashes within the double quotation marks are not considered escape sequences, but rather regular backslash characters. Often this simplifies programming and makes the code easier to read. For example, consider the string “`C:\MyFolder\MySubFolder\MyFile.txt`” with the following assignment:

```
string file = "C:\\MyFolder\\MySubFolder\\MyFile.txt";
```

Using the verbatim string syntax, the assignment can be altered to

```
string file = @"C:\MyFolder\MySubFolder\MyFile.txt";
```

This approach also has the advantage of allowing strings to span multiple lines by preserving all newlines, spaces and tabs.

---

[1] C# provides the `string` keyword as an alias for class `String`. In this book, we use `String` to refer to the class `String` and `string` to refer to an object of class `String`.

15.3 `String` Constructors

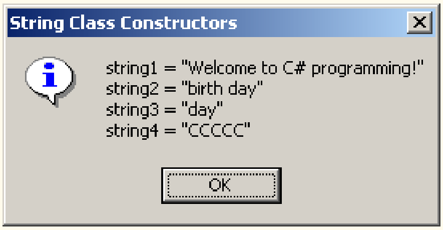
Class **`String`** provides eight constructors for initializing **`string`**s in various ways. Figure 15.1 demonstrates the use of three of the constructors.

Lines 14–16 declare **`string`**s **`output`**, **`originalString`**, **`string1`**, **`string2`**, **`string3`** and **`string4`**. Lines 18–19 allocate the **`char`** array **`characterArray`**, which contains nine characters. Line 22 assigns literal **`string`** **`"Welcome to C# programming!"`** to **`string`** reference **`originalString`**. Line 23 sets **`string1`** to reference the same **`string`** literal.

Line 24 assigns to **`string2`** a new **`string`**, using the **`String`** constructor that takes a character array as an argument. The new **`string`** contains a copy of the characters in array **`characterArray`**.

```
 1  // Fig. 15.1: StringConstructor.cs
 2  // Demonstrating String class constructors.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // test several String class constructors
 8  class StringConstructor
 9  {
10     // The main entry point for the application.
11     [STAThread]
12     static void Main( string[] args )
13     {
14        string output;
15        string originalString, string1, string2,
16           string3, string4;
17
18        char[] characterArray =
19           { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y' };
20
21        // string initialization
22        originalString = "Welcome to C# programming!";
23        string1 = originalString;
24        string2 = new string( characterArray );
25        string3 = new string( characterArray, 6, 3 );
26        string4 = new string( 'C', 5 );
27
28        output = "string1 = " + "\"" + string1 + "\"\n" +
29           "string2 = " + "\"" + string2 + "\"\n" +
30           "string3 = " + "\"" + string3 + "\"\n" +
31           "string4 = " + "\"" + string4 + "\"\n";
32
33        MessageBox.Show( output, "String Class Constructors",
34           MessageBoxButtons.OK, MessageBoxIcon.Information );
35
36     } // end method Main
37
38  } // end class StringConstructor
```

Fig. 15.1 `String` constructors. (Part 1 of 2.)



string1 = "Welcome to C# programming!"
string2 = "birth day"
string3 = "day"
string4 = "CCCCC"

Fig. 15.1 **`String`** constructors. (Part 2 of 2.)

Software Engineering Observation 15.1

In most cases, it is not necessary to make a copy of an existing **`string`***. All* **`string`***s are immutable—their character contents cannot be changed after they are created. Also, if there are one or more references to a* **`string`** *(or any object for that matter), the object cannot be reclaimed by the garbage collector.*

Line 25 assigns to **`string3`** a new **`string`**, using the **`String`** constructor that takes a **`char`** array and two **`int`** arguments. The second argument specifies the starting index position (the *offset*) from which characters in the array are copied. The third argument specifies the number of characters (the *count*) to be copied from the specified starting position in the array. The new **`string`** contains a copy of the specified characters in the array. If the specified offset or count indicates that the program should access an element outside the bounds of the character array, an ***`ArgumentOutOfRangeException`*** is thrown.

Line 26 assigns to **`string4`** a new **`string`**, using the **`String`** constructor that takes as arguments a character and an **`int`** specifying the number of times to repeat that character in the **`string`**.

15.4 `String` Indexer, `Length` Property and `CopyTo` Method

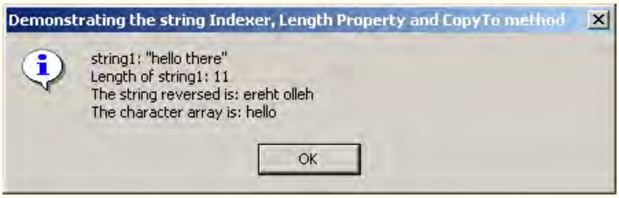
The application in Fig. 15.2 presents the **`String`** indexer, which facilitates the retrieval of any character in the **`string`**, and the **`String`** property ***`Length`***, which returns the length of the **`string`**. The **`String`** method ***`CopyTo`*** copies a specified number of characters from a **`string`** into a **`char`** array.

In this example, we create an application that determines the length of a **`string`**, reverses the order of the characters in the **`string`** and copies a series of characters from the **`string`** into a character array.

```
// Fig. 15.2: StringMethods.cs
// Using the indexer, property Length and method CopyTo
// of class String.

using System;
using System.Windows.Forms;

```

Fig. 15.2 **`String`** indexer, **`Length`** properties and **`CopyTo`** method. (Part 1 of 2.)

```
 8  // creates string objects and displays results of using
 9  // indexer and methods Length and CopyTo
10  class StringMethods
11  {
12     // The main entry point for the application.
13     [STAThread]
14     static void Main( string[] args )
15     {
16        string string1, output;
17        char[] characterArray;
18
19        string1 = "hello there";
20        characterArray = new char[ 5 ];
21
22        // output string
23        output =
24           "string1: \"" + string1 + "\"";
25
26        // test Length property
27        output += "\nLength of string1: " + string1.Length;
28
29        // loop through character in string1 and display
30        // reversed
31        output += "\nThe string reversed is: ";
32
33        for ( int i = string1.Length - 1; i >= 0; i-- )
34           output += string1[ i ];
35
36        // copy characters from string1 into characterArray
37        string1.CopyTo( 0, characterArray, 0, 5 );
38        output += "\nThe character array is: ";
39
40        for ( int i = 0 ; i < characterArray.Length; i++ )
41           output += characterArray[ i ];
42
43        MessageBox.Show( output, "Demonstrating the string " +
44           "Indexer, Length Property and CopyTo method",
45           MessageBoxButtons.OK, MessageBoxIcon.Information );
46
47     } // end method Main
48
49  } // end class StringMethods
```


Demonstrating the string Indexer, Length Property and CopyTo method
OK

```
string1: "hello there"
Length of string1: 11
The string reversed is: ereht olleh
The character array is: hello
```

Fig. 15.2 **`String`** indexer, **`Length`** properties and **`CopyTo`** method. (Part 2 of 2.)

Line 27 uses **`String`** property **`Length`** to determine the number of characters in **`string string1`**. Like arrays, **`string`**s always know their own size.

Lines 33–34 append to `output` the characters of the `string string1` in reverse order. The `string` indexer returns the character at a specific position in the `string`. The `string` indexer treats a `string` as an array of `char`s. The indexer receives an integer argument as the *position number* and returns the character at that position. As with arrays, the first element of a `string` is considered to be at position 0.

Common Programming Error 15.1

Attempting to access a character that is outside the bounds of a `string` *(i.e., an index less than 0 or an index greater than or equal to the* `string`*'s length) results in an* `IndexOutOfRangeException`*.*

Line 37 uses `String` method `CopyTo` to copy the characters of a `string` (`string1`) into a character array (`characterArray`). The first argument given to method `CopyTo` is the index from which the method begins copying characters in the `string`. The second argument is the character array into which the characters are copied. The third argument is the index specifying the location at which the method places the copied characters in the character array. The last argument is the number of characters that the method will copy from the `string`. Lines 40–41 append the `char` array contents to `string output` one character at a time.

15.5 Comparing `Strings`

The next two examples demonstrate the various methods that C# provides for comparing `string`s. To understand how one `string` can be "greater than" or "less than" another `string`, consider the process of alphabetizing a series of last names. The reader would, no doubt, place `"Jones"` before `"Smith"`, because the first letter of `"Jones"` comes before the first letter of `"Smith"` in the alphabet. The alphabet is more than just a set of 26 letters—it is an ordered list of characters in which each letter occurs in a specific position. For example, `Z` is more than just a letter of the alphabet; `Z` is specifically the twenty-sixth letter of the alphabet.

Computers can order characters alphabetically because the characters are represented internally as Unicode numeric codes. When comparing two `string`s, C# simply compares the numeric codes of the characters in the `string`s.

Class `String` provides several ways to compare `string`s. The application in Fig. 15.3 demonstrates the use of method ***`Equals`***, method ***`CompareTo`*** and the equality operator (`==`).

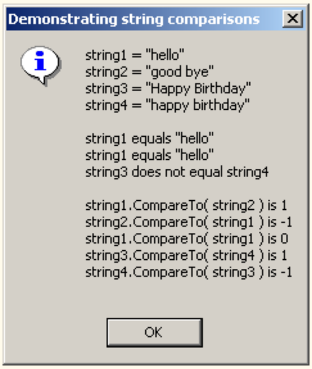
The condition in the `if` structure (line 27) uses instance method `Equals` to compare `string1` and literal `string "hello"` to determine whether they are equal. Method `Equals` (inherited by `String` from class `Object`) tests any two objects for equality (i.e., checks whether the objects contain identical contents). The method returns `true` if the objects are equal and `false` otherwise. In this instance, the preceding condition returns `true`, because `string1` references `string` literal object `"hello"`. Method `Equals` uses a *lexicographical comparison*—the integer Unicode values that represent each character in each `string` are compared. Method `Equals` compares the numeric Unicode values that represent the characters in each `string`. A comparison of the `string "hello"` with the `string "HELLO"` would return `false`, because the numeric representations of lowercase letters are different from the numeric representations of corresponding uppercase letters.

The condition in the second `if` structure (line 33) uses the equality operator (`==`) to compare `string string1` with the literal `string "hello"` for equality. In C#, the equality operator also uses a lexicographical comparison to compare two `string`s. Thus, the condition in the `if` structure evaluates to `true`, because the values of `string1` and `"hello"` are equal. To compare the references of two `string`s, we must explicitly cast the `string`s to type `object` and use the equality operator (`==`).

```
 1  // Fig. 15.3: StringCompare.cs
 2  // Comparing strings.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // compare a number of strings
 8  class StringCompare
 9  {
10     // The main entry point for the application.
11     [STAThread]
12     static void Main( string[] args )
13     {
14        string string1 = "hello";
15        string string2 = "good bye";
16        string string3 = "Happy Birthday";
17        string string4 = "happy birthday";
18        string output;
19
20        // output values of four strings
21        output = "string1 = \"" + string1 + "\"" +
22           "\nstring2 = \"" + string2 + "\"" +
23           "\nstring3 = \"" + string3 + "\"" +
24           "\nstring4 = \"" + string4 + "\"\n\n";
25
26        // test for equality using Equals method
27        if ( string1.Equals( "hello" ) )
28           output += "string1 equals \"hello\"\n";
29        else
30           output += "string1 does not equal \"hello\"\n";
31
32        // test for equality with ==
33        if ( string1 == "hello" )
34           output += "string1 equals \"hello\"\n";
35        else
36           output += "string1 does not equal \"hello\"\n";
37
38        // test for equality comparing case
39        if ( String.Equals( string3, string4 ) )
40           output += "string3 equals string4\n";
41        else
42           output += "string3 does not equal string4\n";
43
```

Fig. 15.3 **`String`** test to determine equality. (Part 1 of 2.)

```
44          // test CompareTo
45          output += "\nstring1.CompareTo( string2 ) is " +
46             string1.CompareTo( string2 ) + "\n" +
47             "string2.CompareTo( string1 ) is " +
48             string2.CompareTo( string1 ) + "\n" +
49             "string1.CompareTo( string1 ) is " +
50             string1.CompareTo( string1 ) + "\n" +
51             "string3.CompareTo( string4 ) is " +
52             string3.CompareTo( string4 ) + "\n" +
53             "string4.CompareTo( string3 ) is " +
54             string4.CompareTo( string3 ) + "\n\n";
55
56          MessageBox.Show( output, "Demonstrating string " +
57             "comparisons", MessageBoxButtons.OK,
58             MessageBoxIcon.Information );
59
60       } // end method Main
61
62    } // end class StringCompare
```



```
string1 = "hello"
string2 = "good bye"
string3 = "Happy Birthday"
string4 = "happy birthday"

string1 equals "hello"
string1 equals "hello"
string3 does not equal string4

string1.CompareTo( string2 ) is 1
string2.CompareTo( string1 ) is -1
string1.CompareTo( string1 ) is 0
string3.CompareTo( string4 ) is 1
string4.CompareTo( string3 ) is -1
```

Fig. 15.3 **`String`** test to determine equality. (Part 2 of 2.)

We present the test for **`string`** equality between **`string3`** and **`string4`** (line 39) to illustrate that comparisons are indeed case sensitive. Here, **`static`** method **`Equals`** (as opposed to the instance method in line 27) is used to compare the values of two **`string`**s. **`"Happy Birthday"`** does not equal **`"happy birthday"`**, so the condition of the **`if`** structure fails, and the message **`"string3 does not equal string4"`** is added to the output message (line 42).

Lines 46–54 use the **`String`** method **`CompareTo`** to compare **`string`**s. Method **`CompareTo`** returns **`0`** if the **`string`**s are equal, a **`-1`** if the **`string`** that invokes **`CompareTo`** is less than the **`string`** that is passed as an argument and a **`1`** if the **`string`** that invokes **`CompareTo`** is greater than the **`string`** that is passed as an argument. Method **`CompareTo`** uses a lexicographical comparison.

Notice that **`CompareTo`** considers **`string3`** to be larger than **`string4`**. The only difference between these two **`string`**s is that **`string3`** contains two uppercase letters. This example illustrates that an uppercase letter has a higher value in the Unicode character set than its corresponding lowercase letter.

The application in Fig. 15.4 shows how to test whether a **string** instance begins or ends with a given **string**. Method ***StartsWith*** determines whether a **string** instance starts with the **string** text passed to it as an argument. Method ***EndsWith*** determines whether a **string** instance ends with the **string** text passed to it as an argument. Application **StringStartEnd**'s **Main** method defines an array of **string**s (called **strings**), which contains **"started"**, **"starting"**, **"ended"** and **"ending"**. The remainder of method **Main** tests the elements of the array to determine whether they start or end with a particular set of characters.

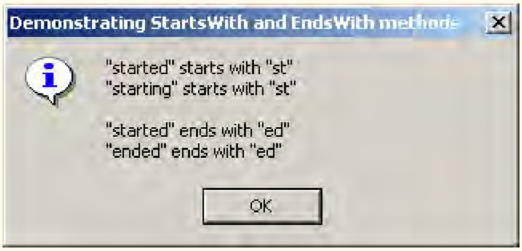
Line 21 uses method **StartsWith**, which takes a **string** argument. The condition in the **if** structure determines whether the **string** at index **i** of the array starts with the characters **"st"**. If so, the method returns **true** and appends **strings[i]** to **string output** for display purposes.

Line 30 uses method **EndsWith**, which also takes a **string** argument. The condition in the **if** structure determines whether the **string** at index **i** of the array ends with the characters **"ed"**. If so, the method returns **true**, and **strings[i]** is appended to **string output** for display purposes.

```
 1  // Fig. 15.4: StringStartEnd.cs
 2  // Demonstrating StartsWith and EndsWith methods.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // testing StartsWith and EndsWith
 8  class StringStartEnd
 9  {
10     // The main entry point for the application.
11     [STAThread]
12     static void Main( string[] args )
13     {
14        string[] strings =
15              { "started", "starting", "ended", "ending" };
16        string output = "";
17
18        //test every string to see if it starts with "st"
19        for ( int i = 0; i < strings.Length; i++ )
20
21           if ( strings[ i ].StartsWith( "st" ) )
22              output += "\"" + strings[ i ] + "\"" +
23                 " starts with \"st\"\n";
24
25        output += "\n";
26
27        // test every string to see if it ends with "ed"
28        for ( int i = 0; i < strings.Length; i ++ )
29
30           if ( strings[ i ].EndsWith( "ed" ) )
31              output += "\"" + strings[ i ] + "\"" +
32                 " ends with \"ed\"\n";
33
```

Fig. 15.4 **StartsWith** and **EndsWith** methods. (Part 1 of 2.)

```
      MessageBox.Show( output, "Demonstrating StartsWith and " +
         "EndsWith methods", MessageBoxButtons.OK,
         MessageBoxIcon.Information );

   } // end method Main

} // end class StringStartEnd
```



"started" starts with "st"
"starting" starts with "st"

"started" ends with "ed"
"ended" ends with "ed"

Fig. 15.4 **`StartsWith`** and **`EndsWith`** methods. (Part 2 of 2.)

15.6 `String` Method `GetHashCode`

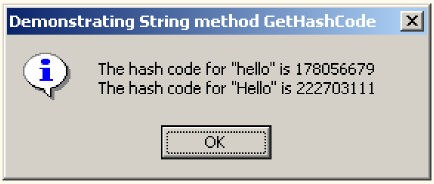
Often, it is necessary to store **`string`**s and other data types in a manner that enables the information to be found quickly. One of the best ways to make information easily accessible is to store it in a hash table. A *hash table* stores an object by performing a special calculation on that object, which produces a *hash code*. The object then is stored at a location in the hash table determined by the calculated hash code. When a program needs to retrieve the information, the same calculation is performed, generating the same hash code. Any object can be stored in a hash table. Class **`Object`** defines method ***`GetHashCode`*** to perform the hash-code calculation. Although all classes inherit this method from class **`Object`**, it is recommended that they override **`Object`**'s default implementation. Class **`String overrides`** method **`GetHashCode`** to provide a good hash-code distribution based on the contents of the **`string`**. We will discuss hashing in detail in Chapter 24, Data Structures.

The example in Fig. 15.5 demonstrates the application of the **`GetHashCode`** method to two **`string`**s (**`"hello"`** and "**`Hello"`**). Here, the hash-code value for each **`string`** is different. However, **`string`**s that are not identical can have the same hash-code value.

```
// Fig. 15.5: StringHashCode.cs
// Demonstrating method GetHashCode of class String.

using System;
using System.Windows.Forms;

```

Fig. 15.5 **`GetHashCode`** method demonstration. (Part 1 of 2.)

```
// testing the GetHashCode method
class StringHashCode
{
   // The main entry point for the application.
   [STAThread]
   static void Main( string[] args )
   {

      string string1 = "hello";
      string string2 = "Hello";
      string output;

      output = "The hash code for \"" + string1 +
         "\" is " + string1.GetHashCode() + "\n";

      output += "The hash code for \"" + string2 +
         "\" is " + string2.GetHashCode() + "\n";

      MessageBox.Show( output, "Demonstrating String " +
         "method GetHashCode", MessageBoxButtons.OK,
         MessageBoxIcon.Information );

   } // end method Main

} // end class StringHashCode
```



```
The hash code for "hello" is 178056679
The hash code for "Hello" is 222703111
```

Fig. 15.5 `GetHashCode` method demonstration. (Part 2 of 2.)

15.7 Locating Characters and Substrings in `Strings`

In many applications, it is necessary to search for a character or set of characters in a **`string`**. For example, a programmer creating a word processor would want to provide capabilities for searching through documents. The application in Fig. 15.6 demonstrates some of the many versions of **`String`** methods ***`IndexOf`***, ***`IndexOfAny`***, ***`LastIndexOf`*** and ***`LastIndexOfAny`***, which search for a specified character or substring in a **`string`**. We perform all searches in this example on the **`string letters`** (initialized with **`"abcdefghijklmabcdefghijklm"`**) located in method **`Main`** of class **`StringIndexMethods`**.

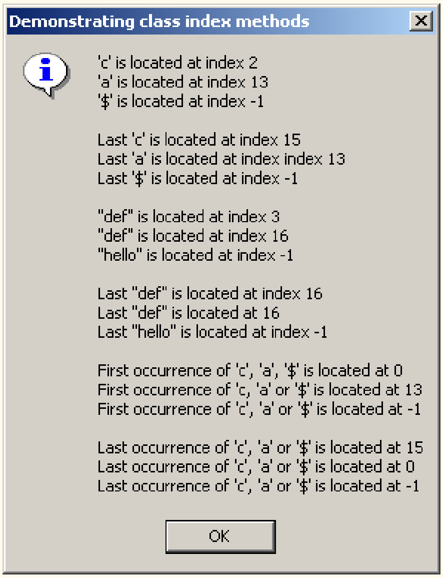
```
// Fig. 15.6: StringIndexMethods.cs
// Using String searching methods.

```

Fig. 15.6 Searching for characters and substrings in `string`s. (Part 1 of 3.)

```
using System;
using System.Windows.Forms;

// testing indexing capabilities of strings
class StringIndexMethods
{
   // The main entry point for the application.
   [STAThread]
   static void Main( string[] args )
   {
      string letters = "abcdefghijklmabcdefghijklm";
      string output = "";
      char[] searchLetters = { 'c', 'a', '$' };

      // test IndexOf to locate a character in a string
      output += "'c' is located at index " +
         letters.IndexOf( 'c' );

      output += "\n'a' is located at index " +
         letters.IndexOf( 'a', 1 );

      output += "\n'$' is located at index " +
         letters.IndexOf( '$', 3, 5 );

      // test LastIndexOf to find a character in a string
      output += "\n\nLast 'c' is located at " +
         "index " + letters.LastIndexOf( 'c' );

      output += "\nLast 'a' is located at index " +
         letters.LastIndexOf( 'a', 25 );

      output += "\nLast '$' is located at index " +
         letters.LastIndexOf( '$', 15, 5 );

      // test IndexOf to locate a substring in a string
      output += "\n\n\"def\" is located at" +
         " index " + letters.IndexOf( "def" );

      output += "\n\"def\" is located at index " +
         letters.IndexOf( "def", 7 );

      output += "\n\"hello\" is located at index " +
         letters.IndexOf( "hello", 5, 15 );

      // test LastIndexOf to find a substring in a string
      output += "\n\nLast \"def\" is located at index " +
         letters.LastIndexOf( "def" );

      output += "\nLast \"def\" is located at " +
         letters.LastIndexOf( "def", 25 );

      output += "\nLast \"hello\" is located at index " +
         letters.LastIndexOf( "hello", 20, 15 );
```

Fig. 15.6 Searching for characters and substrings in `string`s. (Part 2 of 3.)

```

      // test IndexOfAny to find first occurrence of character
      // in array
      output += "\n\nFirst occurrence of 'c', 'a', '$' is " +
         "located at " + letters.IndexOfAny( searchLetters );

      output += "\nFirst occurrence of 'c, 'a' or '$' is " +
         "located at " + letters.IndexOfAny( searchLetters, 7 );

      output += "\nFirst occurrence of 'c', 'a' or '$' is " +
        "located at " + letters.IndexOfAny( searchLetters, 20, 5 );

      // test LastIndexOfAny to find last occurrence of character
      // in array
      output += "\n\nLast occurrence of 'c', 'a' or '$' is " +
         "located at " + letters.LastIndexOfAny( searchLetters );

      output += "\nLast occurrence of 'c', 'a' or '$' is " +
        "located at " + letters.LastIndexOfAny( searchLetters, 1 );

      output += "\nLast occurrence of 'c', 'a' or '$' is " +
         "located at " + letters.LastIndexOfAny(
         searchLetters, 25, 5 );

      MessageBox.Show( output,
         "Demonstrating class index methods",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method Main

} // end class StringIndexMethods
```



```
'c' is located at index 2
'a' is located at index 13
'$' is located at index -1

Last 'c' is located at index 15
Last 'a' is located at index index 13
Last '$' is located at index -1

"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at 16
Last "hello" is located at index -1

First occurrence of 'c', 'a', '$' is located at 0
First occurrence of 'c, 'a' or '$' is located at 13
First occurrence of 'c', 'a' or '$' is located at -1

Last occurrence of 'c', 'a' or '$' is located at 15
Last occurrence of 'c', 'a' or '$' is located at 0
Last occurrence of 'c', 'a' or '$' is located at -1
```

Fig. 15.6 Searching for characters and substrings in `string`s. (Part 3 of 3.)

Lines 20, 23 and 26 use method **`IndexOf`** to locate the first occurrence of a character or substring in a **`string`**. If **`IndexOf`** finds a character, **`IndexOf`** returns the index of the specified character in the **`string`**; otherwise, **`IndexOf`** returns **`-1`**. The expression on line 23 uses a version of method **`IndexOf`** that takes two arguments—the character to search for and the starting index at which the search of the **`string`** should begin. The method does not examine any characters that occur prior to the starting index (in this case **`1`**). The expression in line 26 uses another version of method **`IndexOf`** that takes three arguments—the character to search for, the index at which to start searching and the number of characters to search.

Lines 30, 33 and 36 use method **`LastIndexOf`** to locate the last occurrence of a character in a **`string`**. Method **`LastIndexOf`** performs the search from the end of the **`string`** toward the beginning of the **`string`**. If method **`LastIndexOf`** finds the character, **`LastIndexOf`** returns the index of the specified character in the **`string`**; otherwise, **`LastIndexOf`** returns **`-1`**. There are three versions of **`LastIndexOf`** that search for characters in a **`string`**. The expression in line 30 uses the version of method **`LastIndexOf`** that takes as an argument the character for which to search. The expression in line 33 uses the version of method **`LastIndexOf`** that takes two arguments—the character for which to search and the highest index from which to begin searching backward for the character. The expression in line 36 uses a third version of method **`LastIndexOf`** that takes three arguments—the character for which to search, the starting index from which to start searching backward and the number of characters (the portion of the **`string`**) to search.

Lines 40–56 use versions of **`IndexOf`** and **`LastIndexOf`** that take a **`string`** instead of a character as the first argument. These versions of the methods perform identically to those described above except that they search for sequences of characters (or substrings) that are specified by their **`string`** arguments.

Lines 61–79 use methods **`IndexOfAny`** and **`LastIndexOfAny`**, which take an array of characters as the first argument. These versions of the methods also perform identically to those described above except that they return the index of the first occurrence of any of the characters in the character array argument.

Common Programming Error 15.2

*In the overloaded methods **`LastIndexOf`** and **`LastIndexOfAny`** that take three parameters, the second argument must always be bigger than or equal to the third argument. This might seem counterintuitive, but remember that the search moves from the end of the string toward the start of the string.*

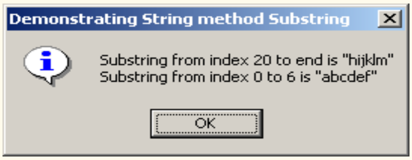
15.8 Extracting Substrings from `Strings`

Class **`String`** provides two ***`Substring`*** methods, which are used to create a new **`string`** by copying part of an existing **`string`**. Each method returns a new **`string`**. The application in Fig. 15.7 demonstrates the use of both methods.

```
// Fig. 15.7: SubString.cs
// Demonstrating the String Substring method.

```

Fig. 15.7 Substrings generated from **`string`**s. (Part 1 of 2.)

```
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // creating substrings
 8  class SubString
 9  {
10     // The main entry point for the application.
11     [STAThread]
12     static void Main( string[] args )
13     {
14        string letters = "abcdefghijklmabcdefghijklm";
15        string output = "";
16
17        // invoke Substring method and pass it one parameter
18        output += "Substring from index 20 to end is \"" +
19           letters.Substring( 20 ) + "\"\n";
20
21        // invoke Substring method and pass it two parameters
22        output += "Substring from index 0 to 6 is \"" +
23           letters.Substring( 0, 6 ) + "\"";
24
25        MessageBox.Show( output,
26           "Demonstrating String method Substring",
27           MessageBoxButtons.OK, MessageBoxIcon.Information );
28
29     } // end method Main
30
31  } // end class SubString
```



```
Substring from index 20 to end is "hijklm"
Substring from index 0 to 6 is "abcdef"
```

Fig. 15.7 Substrings generated from **`string`**s. (Part 2 of 2.)

The statement in line 19 uses the **`Substring`** method that takes one **`int`** argument. The argument specifies the starting index from which the method copies characters in the original **`string`**. The substring returned contains a copy of the characters from the starting index to the end of the **`string`**. If the index specified in the argument is outside the bounds of the **`string`**, the program throws an **`ArgumentOutOfRangeException`**.

The second version of method **`Substring`** (line 23) takes two **`int`** arguments. The first argument specifies the starting index from which the method copies characters from the original **`string`**. The second argument specifies the length of the substring to be copied. The substring returned contains a copy of the specified characters from the original **`string`**.

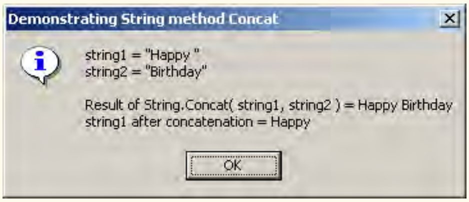
15.9 Concatenating `String`s

The **+** operator (discussed in Chapter 3, Introduction to C# Programming) is not the only way to perform **`string`** concatenation. The **`static`** method ***`Concat`*** of class **`String`** (Fig. 15.8) concatenates two **`string`**s and returns a new **`string`** containing the combined characters from both original **`string`**s. Line 23 appends the characters from **`string2`** to the end of **`string1`**, using method **`Concat`**. The statement on line 23 does not modify the original **`string`**s.

15.10 Miscellaneous `String` Methods

Class **`String`** provides several methods that return modified copies of **`string`**s. The application in Fig. 15.9 demonstrates the use of these methods, which include **`String`** methods ***`Replace`***, ***`ToLower`***, ***`ToUpper`***, ***`Trim`*** and ***`ToString`***.

```
 1  // Fig. 15.8: SubConcatination.cs
 2  // Demonstrating String class Concat method.
 3
 4  using System;
 5  using System.Windows.Forms;
 6
 7  // concatenates strings using String method Concat
 8  class StringConcatenation
 9  {
10     // The main entry point for the application.
11     [STAThread]
12     static void Main( string[] args )
13     {
14        string string1 = "Happy ";
15        string string2 = "Birthday";
16        string output;
17
18        output = "string1 = \"" + string1 + "\"\n" +
19           "string2 = \"" + string2 + "\"";
20
21        output +=
22           "\n\nResult of String.Concat( string1, string2 ) = " +
23           String.Concat( string1, string2 );
24
25        output += "\nstring1 after concatenation = " + string1;
26
27        MessageBox.Show( output,
28           "Demonstrating String method Concat",
29           MessageBoxButtons.OK, MessageBoxIcon.Information );
30
31     } // end method Main
32
33  } // end class StringConcatenation
```



```
string1 = "Happy "
string2 = "Birthday"

Result of String.Concat( string1, string2 ) = Happy Birthday
string1 after concatenation = Happy
```

Fig. 15.8 **`Concat static`** method.

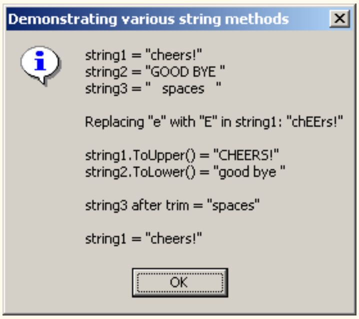
Line 27 uses **`String`** method **`Replace`** to return a new **`string`**, replacing every occurrence in **`string1`** of character **`'e'`** with character **`'E'`**. Method **`Replace`** takes two arguments—a **`string`** for which to search and another **`string`** with which to replace all matching occurrences of the first argument. The original **`string`** remains unchanged. If there are no occurrences of the first argument in the **`string`**, the method returns the original **`string`**.

`String` method **`ToUpper`** generates a new **`string`** (line 31) that replaces any lowercase letters in **`string1`** with their uppercase equivalent. The method returns a new **`string`** containing the converted **`string`**; the original **`string`** remains unchanged. If there are no characters to convert to uppercase, the method returns the original **`string`**. Line 32 uses **`String`** method **`ToLower`** to return a new **`string`** in which any uppercase letters in **`string1`** are replaced by their lowercase equivalents. The original **`string`** is unchanged. As with **`ToUpper`**, if there are no characters to convert to lowercase, method **`ToLower`** returns the original **`string`**.

```
1  // Fig. 15.9: StringMiscellaneous2.cs
2  // Demonstrating String methods Replace, ToLower, ToUpper, Trim
3  // and ToString.
4
5  using System;
6  using System.Windows.Forms;
7
8  // creates strings using methods Replace, ToLower, ToUpper, Trim
9  class StringMethods2
10 {
11    // The main entry point for the application.
12    [STAThread]
13    static void Main( string[] args )
14    {
15       string string1 = "cheers!";
16       string string2 = "GOOD BYE ";
17       string string3 = "   spaces   ";
18       string output;
19
20       output = "string1 = \"" + string1 + "\"\n" +
21          "string2 = \"" + string2 + "\"\n" +
22          "string3 = \"" + string3 + "\"";
23
24       // call method Replace
25       output +=
26          "\n\nReplacing \"e\" with \"E\" in string1: \"" +
27          string1.Replace( 'e', 'E' ) + "\"";
28
29       // call ToLower and ToUpper
30       output += "\n\nstring1.ToUpper() = \"" +
31          string1.ToUpper() + "\"\nstring2.ToLower() = \"" +
32          string2.ToLower() + "\"";
33
```

Fig. 15.9 **`String`** methods **`Replace`**, **`ToLower`**, **`ToUpper`**, **`Trim`** and **`ToString`**. (Part 1 of 2.)

```
34        // call Trim method
35        output += "\n\nstring3 after trim = \"" +
36           string3.Trim() + "\"";
37
38        // call ToString method
39        output += "\n\nstring1 = \"" + string1.ToString() + "\"";
40
41        MessageBox.Show( output,
42           "Demonstrating various string methods",
43           MessageBoxButtons.OK, MessageBoxIcon.Information );
44
45     } // end method Main
46
47  } // end class StringMethods2
```



```
string1 = "cheers!"
string2 = "GOOD BYE "
string3 = "  spaces  "

Replacing "e" with "E" in string1: "chEErs!"

string1.ToUpper() = "CHEERS!"
string2.ToLower() = "good bye "

string3 after trim = "spaces"

string1 = "cheers!"
```

Fig. 15.9 **`String`** methods **`Replace`**, **`ToLower`**, **`ToUpper`**, **`Trim`** and **`ToString`**. (Part 2 of 2.)

Line 36 uses **`String`** method **`Trim`** to remove all whitespace characters that appear at the beginning and end of a **`string`**. Without otherwise altering the original **`string`**, the method returns a new **`string`** that contains the **`string`**, but omits leading or trailing whitespace characters. Another version of method **`Trim`** takes a character array and returns a **`string`** that does not contain the characters in the array argument.

Line 39 uses class **`String`**'s method **`ToString`** to show that the various other methods employed in this application have not modified **`string1`**. Why is the **`ToString`** method provided for class **`String`**? In C#, all objects are derived from class **`Object`**, which defines **`virtual`** method **`ToString`**. Thus, method **`ToString`** can be called to obtain a **`string`** representation of any object. If a class that inherits from **`Object`** (such as **`String`**) does not override method **`ToString`**, the class uses the default version from class **`Object`**, which returns a **`string`** consisting of the object's class name. Classes usually override method **`ToString`** to express the contents of an object as text. Class **`String`** overrides method **`ToString`** so that, instead of returning the class name, it simply returns the **`string`**.

15.11 Class `StringBuilder`

The **`String`** class provides many capabilities for processing **`string`**s. However a **`string`**'s contents can never change. Operations that seem to concatenate **`string`**s are in fact assigning **`string`** references to newly created **`string`**s (e.g., the **`+=`** operator creates a new **`string`** and assigns the initial **`string`** reference to the newly created **`string`**).

The next several sections discuss the features of class **`StringBuilder`** (namespace **`System.Text`**), used to create and manipulate dynamic string information—i.e., mutable strings. Every **`StringBuilder`** can store a certain number of characters that is specified by its capacity. Exceeding the capacity of a **`StringBuilder`** causes the capacity to expand to accommodate the additional characters. As we will see, members of class **`StringBuilder`**, such as methods **`Append`** and **`AppendFormat`**, can be used for concatenation like the operators **`+`** and **`+=`** for class **`String`**.

Software Engineering Observation 15.2

*Objects of class **`String`** are constant strings, whereas object of class **`StringBuilder`** are mutable strings. C# can perform certain optimizations involving **`string`**s (such as the sharing of one **`string`** among multiple references), because it knows these objects will not change.*

Performance Tip 15.2

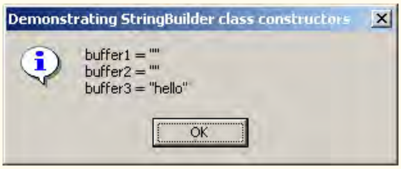
*When given the choice between using a **`string`** to represent a string and using a **`StringBuilder`** object to represent that string, always use a **`string`** if the contents of the object will not change. When appropriate, using **`string`**s instead of **`StringBuilder`** objects improves performance.*

Class **`StringBuilder`** provides six overloaded constructors. Class **`StringBuilderConstructor`** (Fig. 15.10) demonstrates the use of three of these overloaded constructors.

```
// Fig. 15.10: StringBuilderConstructor.cs
// Demonstrating StringBuilder class constructors.

using System;
using System.Windows.Forms;
using System.Text;

// creates three StringBuilder with three constructors
class StringBuilderConstructor
{
   // The main entry point for the application.
   [STAThread]
   static void Main( string[] args )
   {
      StringBuilder buffer1, buffer2, buffer3;
      string output;

      buffer1 = new StringBuilder();
      buffer2 = new StringBuilder( 10 );
      buffer3 = new StringBuilder( "hello" );

      output = "buffer1 = \"" + buffer1.ToString() + "\"\n";

      output += "buffer2 = \"" + buffer2.ToString() + "\"\n";

```

Fig. 15.10 **`StringBuilder`** class constructors. (Part 1 of 2.)

```
26          output += "buffer3 = \"" + buffer3.ToString() + "\"\n";
27
28          MessageBox.Show( output,
29             "Demonstrating StringBuilder class constructors",
30             MessageBoxButtons.OK, MessageBoxIcon.Information );
31
32       } // end method Main
33
34    } // end class StringBuilderConstructor
```



Demonstrating StringBuilder class constructors

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

Fig. 15.10 **`StringBuilder`** class constructors. (Part 2 of 2.)

Line 18 employs the no-argument **`StringBuilder`** constructor to create a **`StringBuilder`** that contains no characters and has a default initial capacity of 16 characters. Line 19 uses the **`StringBuilder`** constructor that takes an **`int`** argument to create a **`StringBuilder`** that contains no characters and has the initial capacity specified in the **`int`** argument (i.e., **`10`**). Line 20 uses the **`StringBuilder`** constructor that takes a **`string`** argument to create a **`StringBuilder`** containing the characters of the **`string`** argument. The initial capacity is the smallest power of two greater than the number of characters in the **`string`** passed as an argument.

Lines 22–26 use **`StringBuilder`** method **`ToString`** to obtain a **`string`** representation of the **`StringBuilders`**’ contents. This method returns the **`StringBuilders`**’ underlying string.

15.12 `StringBuilder` Indexer, `Length` and `Capacity` Properties, and `EnsureCapacity` Method

Class **`StringBuilder`** provides the ***`Length`*** and ***`Capacity`*** properties to return the number of characters currently in a **`StringBuilder`** and the number of characters that a **`StringBuilder`** can store without allocating more memory, respectively. These properties also can increase or decrease the length or the capacity of the **`StringBuilder`**.

Method ***`EnsureCapacity`*** allows programmers to guarantee that a **`StringBuilder`** has a capacity that reduces the number of times the capacity must be increased. Method **`EnsureCapacity`** doubles the **`StringBuilder`** instance’s current capacity. If this doubled value is greater than the value that the programmer wishes to ensure, it becomes the new capacity. Otherwise, **`EnsureCapacity`** alters the capacity to make it one more than the requested number. For example, if the current capacity is 17 and we wish to make it 40, 17 multiplied by 2 is not greater than 40, so the call will result in a new capacity of 41. If the current capacity is 23 and we wish to make it 40, 23 will be multiplied by 2 to result in a new capacity of 46. Both 41 and 46 are greater than 40, and so a capacity of 40 is indeed ensured by method **`EnsureCapacity`**. The program in Fig. 15.11 demonstrates the use of these methods and properties.

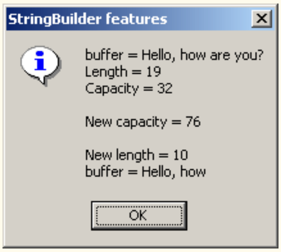
The program contains one **`StringBuilder`**, called **`buffer`**. Lines 15–16 of the program use the **`StringBuilder`** constructor that takes a **`string`** argument to instantiate the **`StringBuilder`** and initialize its value to **`"Hello, how are you?"`**. Lines 19–21 append to **`output`** the content, length and capacity of the **`StringBuilder`**. In the output window, notice that the capacity of the **`StringBuilder`** is initially 32. Remember, the **`StringBuilder`** constructor that takes a **`string`** argument creates a **`StringBuilder`** object with an initial capacity that is the smallest power of two greater than the number of characters in the **`string`** passed as an argument.

Line 24 expands the capacity of the **`StringBuilder`** to a minimum of 75 characters. The current capacity (**`32`**) multiplied by two is less than 75, so method **`EnsureCapacity`** increases the capacity to one greater than 75 (i.e., 76). If new characters are added to a **`StringBuilder`** so that its length exceeds its capacity, the capacity grows to accommodate the additional characters in the same manner as if method **`EnsureCapacity`** had been called.

```
1  // Fig. 15.11: StringBuilderFeatures.cs
2  // Demonstrating some features of class StringBuilder.
3
4  using System;
5  using System.Windows.Forms;
6  using System.Text;
7
8  // uses some of class StringBuilder's methods
9  class StringBuilderFeatures
10 {
11    // The main entry point for the application.
12    [STAThread]
13    static void Main( string[] args )
14    {
15       StringBuilder buffer =
16          new StringBuilder( "Hello, how are you?" );
17
18       // use Length and Capacity properties
19       string output = "buffer = " + buffer.ToString() +
20          "\nLength = " + buffer.Length +
21          "\nCapacity = " + buffer.Capacity;
22
23       // use EnsureCapacity method
24       buffer.EnsureCapacity( 75 );
25
26       output += "\n\nNew capacity = " +
27          buffer.Capacity;
28
29       // truncate StringBuilder by setting Length property
30       buffer.Length = 10;
31
32       output += "\n\nNew length = " +
33          buffer.Length + "\nbuffer = ";
34
```

Fig. 15.11 **`StringBuilder`** size manipulation. (Part 1 of 2.)

```
35          // use StringBuilder indexer
36          for ( int i = 0; i < buffer.Length; i++ )
37             output += buffer[ i ];
38
39          MessageBox.Show( output, "StringBuilder features",
40             MessageBoxButtons.OK, MessageBoxIcon.Information );
41
42       } // end method Main
43
44    } // end class StringBuilderFeatures
```


StringBuilder features

```
buffer = Hello, how are you?
Length = 19
Capacity = 32

New capacity = 76

New length = 10
buffer = Hello, how
```

Fig. 15.11 `StringBuilder` size manipulation. (Part 2 of 2.)

Line 30 uses **`Length`**’s **`Set`** accessor to set the length of the **`StringBuilder`** to **`10`**. If the specified length is less than the current number of characters in the **`StringBuilder`**, the contents of **`StringBuilder`** are truncated to the specified length (i.e., the program discards all characters in the **`StringBuilder`** that occur after the specified length). If the specified length is greater than the number of characters currently in the **`StringBuilder`**, null characters (characters with the numeric representation **`0`** that signal the end of a **`string`**) are appended to the **`StringBuilder`** until the total number of characters in the **`StringBuilder`** is equal to the specified length.

Common Programming Error 15.3

*Assigning **`null`** to a **`string`** reference can lead to logic errors. The keyword **`null`** is a null reference, not a **`string`**. Do not confuse **`null`** with the empty string, **`""`** (the **`string`** that is of length 0 and contains no characters).*

15.13 `StringBuilder` `Append` and `AppendFormat` Methods

Class **`StringBuilder`** provides 19 overloaded ***`Append`*** methods that allow various data-type values to be added to the end of a **`StringBuilder`**. C# provides versions for each of the primitive data types and for character arrays, **`String`**s and **`Object`**s. (Remember that method **`ToString`** produces a **`string`** representation of any **`Object`**.) Each of the methods takes an argument, converts it to a **`string`** and appends it to the **`StringBuilder`**. Figure 15.12 demonstrates the use of several **`Append`** methods.

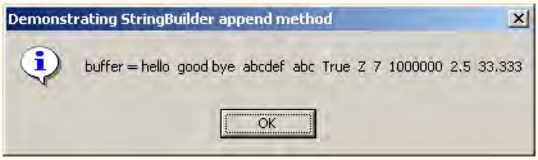
Lines 29–47 use 10 different overloaded **`Append`** methods to attach the objects created in lines 15–26 to the end of the **`StringBuilder`**. **`Append`** behaves similarly to the **`+`** operator which is used with **`string`**s. Just as **`+`** seems to append objects to a **`string`**, method **`Append`** can append data types to a **`StringBuilder`**’s underlying string.

Class **`StringBuilder`** also provides method ***`AppendFormat`***, which converts a **`string`** to a specified format and then appends it to the **`StringBuilder`**. The example in Fig. 15.13 demonstrates the use of this method.

```
// Fig. 15.12: StringBuilderAppend.cs
// Demonstrating StringBuilder Append methods.

using System;
using System.Windows.Forms;
using System.Text;

// testing the Append method
class StringBuilderAppend
{
   // The main entry point for the application.
   [STAThread]
   static void Main( string[] args )
   {
      object objectValue = "hello";
      string stringValue = "good bye";
      char[] characterArray = { 'a', 'b', 'c', 'd',
                                'e', 'f' };

      bool booleanValue = true;
      char characterValue = 'Z';
      int integerValue = 7;
      long longValue = 1000000;
      float floatValue = 2.5F;
      double doubleValue = 33.333;
      StringBuilder buffer = new StringBuilder();

      // use method Append to append values to buffer
      buffer.Append( objectValue );
      buffer.Append( "  " );
      buffer.Append( stringValue );
      buffer.Append( "  " );
      buffer.Append( characterArray );
      buffer.Append( "  " );
      buffer.Append( characterArray, 0, 3 );
      buffer.Append( "  " );
      buffer.Append( booleanValue );
      buffer.Append( "  " );
      buffer.Append( characterValue );
      buffer.Append( "  " );
      buffer.Append( integerValue );
      buffer.Append( "  " );
      buffer.Append( longValue );
      buffer.Append( "  " );
      buffer.Append( floatValue );
      buffer.Append( "  " );
      buffer.Append( doubleValue );

```

Fig. 15.12 Append methods of **`StringBuilder`**. (Part 1 of 2.)

```
      MessageBox.Show( "buffer = " + buffer.ToString(),
         "Demonstrating StringBuilder append method",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method Main

} // end class StringBuilderAppend
```



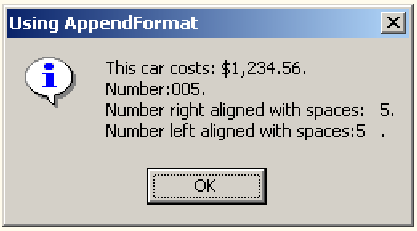
buffer = hello good bye abcdef abc True Z 7 1000000 2.5 33.333

Fig. 15.12 Append methods of **`StringBuilder`**. (Part 2 of 2.)

```
// Fig. 15.13: StringBuilderAppendFormat.cs
// Demonstrating method AppendFormat.

using System;
using System.Windows.Forms;
using System.Text;

// use the AppendFormat method
class StringBuilderAppendFormat
{
   // The main entry point for the application.
   [STAThread]
   static void Main( string[] args )
   {
      StringBuilder buffer = new StringBuilder();
      string string1, string2;

      // formatted string
      string1 = "This {0} costs: {1:C}.\n";

      // string1 argument array
      object[] objectArray = new object[ 2 ];

      objectArray[ 0 ] = "car";
      objectArray[ 1 ] = 1234.56;

      // append to buffer formatted string with argument
      buffer.AppendFormat( string1, objectArray );

      // formatted string
      string2 = "Number:{0:d3}.\n" +
         "Number right aligned with spaces:{0, 4}.\n" +
         "Number left aligned with spaces:{0, -4}.";

```

Fig. 15.13 **`StringBuilder`**'s **`AppendFormat`** method. (Part 1 of 2.)

```
35           // append to buffer formatted string with argument
36           buffer.AppendFormat( string2, 5 );
37
38           // display formatted strings
39           MessageBox.Show( buffer.ToString(), "Using AppendFormat",
40              MessageBoxButtons.OK, MessageBoxIcon.Information );
41
42        } // end method Main
43
44     } // end class StringBuilderAppendFormat
```



```
This car costs: $1,234.56.
Number:005.
Number right aligned with spaces:    5.
Number left aligned with spaces:5    .
```

Fig. 15.13 **`StringBuilder`**'s **`AppendFormat`** method. (Part 2 of 2.)

Line 19 creates a **`string`** that contains formatting information. The information enclosed within the braces determines how to format a specific piece of information. Formats have the form **`{X[,Y][:FormatString]}`**, where **`X`** is the number of the argument to be formatted, counting from zero. **`Y`** is an optional argument, which can be positive or negative, indicating how many characters should be in the result of formatting. If the resulting **`string`** is less than the number **`Y`**, the **`string`** will be padded with spaces to make up for the difference. A positive integer aligns the **`string`** to the right; a negative integer aligns it to the left. The optional **`FormatString`** applies a particular format to the argument: Currency, decimal or scientific, among others. In this case, "**`{0}`**" means the first argument will be printed out. "**`{1:C}`**" specifies that the second argument will be formatted as a currency value.

Line 28 shows a version of **`AppendFormat`**, which takes two parameters—a **`string`** specifying the format and an array of objects to serve as the arguments to the format **`string`**. The argument referred to by "**`{0}`**" is in the object array at index **`0`**, and so on.

Lines 31–33 define another **`string`** used for formatting. The first format "**`{0:D3}`**" specifies that the first argument will be formatted as a three-digit decimal, meaning any number that has fewer than three digits will have leading zeros placed in front to make up the difference. The next format, "**`{0, 4}`**" specifies that the formatted **`string`** should have four characters and should be right aligned. The third format, "**`{0, -4}`**" specifies that the **`string`**s should be aligned to the left. For more formatting options, please refer to the documentation.

Line 36 uses a version of **`AppendFormat`** that takes two parameters: A **`string`** containing a format and an object to which the format is applied. In this case, the object is the number **`5`**. The output of Fig. 15.13 displays the result of applying these two version of **`AppendFormat`** with their respective arguments.

15.14 StringBuilder Insert, Remove and Replace Methods

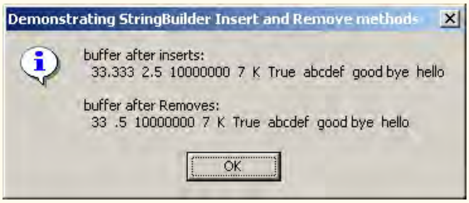
Class **StringBuilder** provides 18 overloaded ***Insert*** methods to allow various data-type values to be inserted at any position in a **StringBuilder**. The class provides versions for each of the primitive data types and for character arrays, **String**s and **Object**s. (Remember that method **ToString** produces a **string** representation of any **Object**.) Each method takes its second argument, converts it to a **string** and inserts the **string** into the **StringBuilder** in front of the index specified by the first argument. The index specified by the first argument must be greater than or equal to **0** and less than the length of the **StringBuilder**; otherwise, the program throws an **ArgumentOutOfRangeException**.

Class **StringBuilder** also provides method ***Remove*** for deleting any portion of a **StringBuilder**. Method **Remove** takes two arguments—the index at which to begin deletion and the number of characters to delete. The sum of the starting subscript and the number of characters to be deleted must always be less than the length of the **StringBuilder**; otherwise, the program throws an **ArgumentOutOfRangeException**. The **Insert** and **Remove** methods are demonstrated in Fig. 15.14.

```
// Fig. 15.14: StringBuilderInsertRemove.cs
// Demonstrating methods Insert and Remove of the
// StringBuilder class.

using System;
using System.Windows.Forms;
using System.Text;

// test the Insert and Remove methods
class StringBuilderInsertRemove
{
   // The main entry point for the application.
   [STAThread]
   static void Main( string[] args )
   {
      object objectValue = "hello";
      string stringValue = "good bye";
      char[] characterArray = { 'a', 'b', 'c',
                                'd', 'e', 'f' };

      bool booleanValue = true;
      char characterValue = 'K';
      int integerValue = 7;
      long longValue = 10000000;
      float floatValue = 2.5F;
      double doubleValue = 33.333;
      StringBuilder buffer = new StringBuilder();
      string output;

      // insert values into buffer
      buffer.Insert(0, objectValue);
```

Fig. 15.14 **StringBuilder** text insertion and removal. (Part 1 of 2.)

```
      buffer.Insert(0, "  ");
      buffer.Insert(0, stringValue);
      buffer.Insert(0, "  ");
      buffer.Insert(0, characterArray);
      buffer.Insert(0, "  ");
      buffer.Insert(0, booleanValue);
      buffer.Insert(0, "  ");
      buffer.Insert(0, characterValue);
      buffer.Insert(0, "  ");
      buffer.Insert(0, integerValue);
      buffer.Insert(0, "  ");
      buffer.Insert(0, longValue);
      buffer.Insert(0, "  ");
      buffer.Insert(0, floatValue);
      buffer.Insert(0, "  ");
      buffer.Insert(0, doubleValue);
      buffer.Insert(0, "  ");

      output = "buffer after inserts: \n" +
         buffer.ToString() + "\n\n";

      buffer.Remove( 10, 1 ); // delete 2 in 2.5
      buffer.Remove( 2, 4 );  // delete .333 in 33.333

      output += "buffer after Removes:\n" +
         buffer.ToString();

      MessageBox.Show( output, "Demonstrating StringBuilder " +
         "Insert and Remove methods", MessageBoxButtons.OK,
         MessageBoxIcon.Information );

   } // end method Main

} // end class StringBuilderInsertRemove
```



```
buffer after inserts:
 33.333 2.5 10000000 7 K True abcdef good bye hello

buffer after Removes:
 33 .5 10000000 7 K True abcdef good bye hello
```

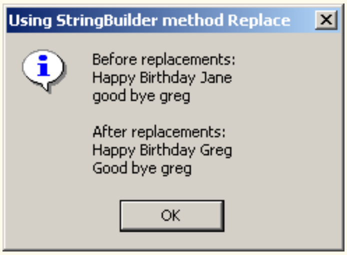
Fig. 15.14 **`StringBuilder`** text insertion and removal. (Part 2 of 2.)

Another useful method included with **`StringBuilder`** is **`Replace`**. **`Replace`** searches for a specified **`string`** or character and substitutes another **`string`** or character in its place. Figure 15.15 demonstrates this method.

```
// Fig. 15.15: StringBuilderReplace.cs
// Demonstrating method Replace.

```

Fig. 15.15 **`StringBuilder`** text replacement. (Part 1 of 2.)

```
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Text;
 7
 8  // testing the Replace method
 9  class StringBuilderReplace
10  {
11     // The main entry point for the application.
12     [STAThread]
13     static void Main( string[] args )
14     {
15        StringBuilder builder1 =
16           new StringBuilder( "Happy Birthday Jane" );
17
18        StringBuilder builder2 =
19           new StringBuilder( "good bye greg" );
20
21        string output = "Before replacements:\n" +
22           builder1.ToString() + "\n" + builder2.ToString();
23
24        builder1.Replace( "Jane", "Greg" );
25        builder2.Replace( 'g', 'G', 0, 5 );
26
27        output += "\n\nAfter replacements:\n" +
28           builder1.ToString() + "\n" + builder2.ToString();
29
30        MessageBox.Show( output,
31           "Using StringBuilder method Replace",
32           MessageBoxButtons.OK, MessageBoxIcon.Information );
33
34     } // end method Main
35
36  } // end class StringBuilderReplace
```



```
Before replacements:
Happy Birthday Jane
good bye greg

After replacements:
Happy Birthday Greg
Good bye greg
```

Fig. 15.15 `StringBuilder` text replacement. (Part 2 of 2.)

Line 24 uses method **`Replace`** to replace all instances of the **`string "Jane"`** with the **`string "Greg"`** in **`builder1`**. Another overload of this method takes two characters as parameters and replaces each occurrence of the first with one of the second. Line 25 uses an overload of **`Replace`** that takes four parameters, the first two of which are characters and the second two of which are **`int`**s. The method replaces all instances of the first character with the second, beginning at the index specified by the first **`int`** and continuing for a count specified by the second. Thus, in this case, **`Replace`** looks through only five characters starting with the character at index **`0`**. As the outputs illustrates, this version of

`Replace` replaces `g` with `G` in the word `"good"`, but not in `"greg"`. This is because the `g`s in `"greg"` do not fall in the range indicated by the `int` arguments (i.e., between indexes `0` and `4`).

15.15 Char Methods

C# provides a data type, called a *structure*, that is similar to a class. Although structures and classes are comparable in many ways, structures are a value type. Like classes, structures include methods and properties. Both use the same modifiers (such as `public`, `private` and `protected`) and access members via the member access operator (`.`). However, classes are created by using the keyword `class`, but structures are created using the keyword *`struct`*.

Many of the primitive data types that we have used in this book are actually aliases for different structures. For instance, an `int` is defined by structure `System.Int32`, a `Long` by `System.Int64`, and so on. These structures are derived from class *`ValueType`*, which in turn is derived from class `Object`. In this section, we present structure *`Char`*, which is the structure for characters.

Most `Char` methods are `static`, take at least one character argument and perform either a test or a manipulation on the character. We present several of these methods in the next example. Figure 15.16 demonstrates `static` methods that test characters to determine whether they are of a specific character type and `static` methods that perform case conversions on characters.

This Windows application contains a prompt, a `TextBox` into which the user can input a character, a button that the user can press after entering a character and a second `TextBox` that displays the output of our analysis. When the user clicks the **Analyze Character** button, event handler `analyzeButton_Click` (lines 32–37) is invoked. This method converts the entered data from a `string` to a `Char`, using method `Convert.ToChar` (line 35). On line 36, we call method `BuildOutput`, which is defined in lines 40–72.

Line 45 uses `Char` method *`IsDigit`* to determine whether character `inputCharacter` is defined as a digit. If so, the method returns `true`; otherwise, it returns `false`.

Line 48 uses `Char` method *`IsLetter`* to determine whether character `inputCharacter` is a letter. If so, the method returns `true`; otherwise, it returns `false`. Line 51 uses `Char` method *`IsLetterOrDigit`* to determine whether character `inputCharacter` is a letter or a digit. If so, the method returns `true`; otherwise, it returns `false`.

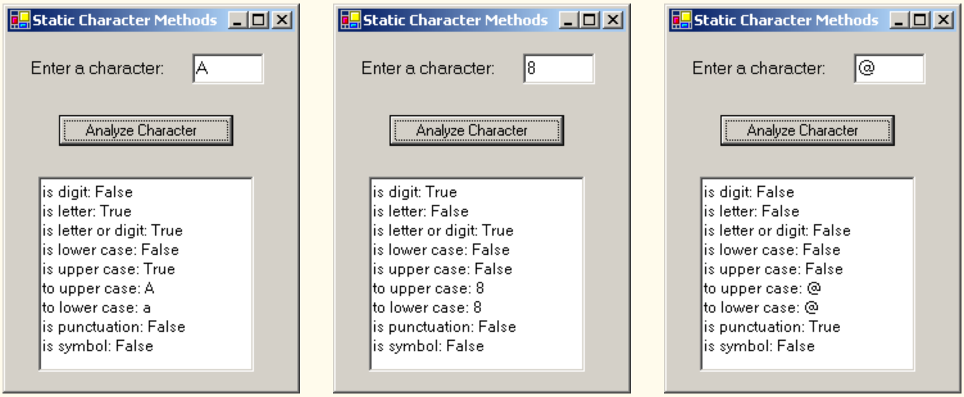
```
1  // Fig. 15.16: CharMethods.cs
2  // Demonstrates static character testing methods
3  // from Char structure
4
5  using System;
6  using System.Drawing;
7  using System.Collections;
8  using System.ComponentModel;
9  using System.Windows.Forms;
10 using System.Data;
```

Fig. 15.16 `Char`'s `static` character-testing methods and case-conversion methods. (Part 1 of 3.)

```

// Form displays information about specific characters.
public class StaticCharMethods : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label enterLabel;
   private System.Windows.Forms.TextBox inputTextBox;
   private System.Windows.Forms.Button analyzeButton;
   private System.Windows.Forms.TextBox outputTextBox;

   private System.ComponentModel.Container components = null;

   // The main entry point for the application.
   [STAThread]
   static void Main()
   {
      Application.Run( new StaticCharMethods() );
   }

   // Visual Studio .NET generated code

   // handle analyzeButton_Click
   private void analyzeButton_Click(
      object sender, System.EventArgs e )
   {
      char character = Convert.ToChar( inputTextBox.Text );
      BuildOutput( character );
   }

   // display character information in outputTextBox
   private void BuildOutput( char inputCharacter )
   {
      string output;

      output = "is digit: " +
         Char.IsDigit( inputCharacter ) + "\r\n";

      output += "is letter: " +
         Char.IsLetter( inputCharacter ) + "\r\n";

      output += "is letter or digit: " +
         Char.IsLetterOrDigit( inputCharacter ) + "\r\n";

      output += "is lower case: " +
         Char.IsLower( inputCharacter ) + "\r\n";

      output += "is upper case: " +
         Char.IsUpper( inputCharacter ) + "\r\n";

      output += "to upper case: " +
         Char.ToUpper( inputCharacter ) + "\r\n";

```

Fig. 15.16 **`Char`**'s **`static`** character-testing methods and case-conversion methods. (Part 2 of 3.)

```
62          output += "to lower case: " +
63             Char.ToLower( inputCharacter ) + "\r\n";
64
65          output += "is punctuation: " +
66             Char.IsPunctuation( inputCharacter ) + "\r\n";
67
68          output += "is symbol: " + Char.IsSymbol( inputCharacter );
69
70          outputTextBox.Text = output;
71
72       } // end method BuildOutput
73
74    } // end class StaticCharMethods
```



```
is digit: False
is letter: True
is letter or digit: True
is lower case: False
is upper case: True
to upper case: A
to lower case: a
is punctuation: False
is symbol: False
```

```
is digit: True
is letter: False
is letter or digit: True
is lower case: False
is upper case: False
to upper case: 8
to lower case: 8
is punctuation: False
is symbol: False
```

```
is digit: False
is letter: False
is letter or digit: False
is lower case: False
is upper case: False
to upper case: @
to lower case: @
is punctuation: True
is symbol: False
```

Fig. 15.16 `Char`'s `static` character-testing methods and case-conversion methods. (Part 3 of 3.)

Line 54 uses **`Char`** method ***`IsLower`*** to determine whether character **`inputCharacter`** is a lowercase letter. If so, the method returns **`true`**; otherwise, it returns **`false`**. Line 57 uses **`Char`** method ***`IsUpper`*** to determine whether character **`inputCharacter`** is an uppercase letter. If so, the method returns **`true`**; otherwise, it returns **`false`**. Line 60 uses **`Char`** method ***`ToUpper`*** to convert the character **`inputCharacter`** to its uppercase equivalent. The method returns the converted character if the character has an uppercase equivalent; otherwise, the method returns its original argument. Line 63 uses **`Char`** method ***`ToLower`*** to convert the character **`inputCharacter`** to its lowercase equivalent. The method returns the converted character if the character has a lowercase equivalent; otherwise, the method returns its original argument.

Line 66 uses **`Char`** method ***`IsPunctuation`*** to determine whether character **`inputCharacter`** is a punctuation mark. If so, the method returns **`true`**; otherwise, it returns **`false`**. Line 68 uses **`Char`** method ***`IsSymbol`*** to determine whether character **`inputCharacter`** is a symbol. If so, the method returns **`true`**; otherwise it returns **`false`**.

Structure type **`Char`** also contains other methods not shown in this example. Many of the **`static`** methods are similar; for instance, ***`IsWhiteSpace`*** is used to determine whether a certain character is a whitespace character (e.g., newline, tab or space). The structure also contains several **`public`** instance methods; many of these, such as methods

`ToString` and `Equals`, are methods that we have seen before in other classes. This group includes method *`CompareTo`*, which is used to compare two character values with one another.

15.16 Card Shuffling and Dealing Simulation

In this section, we use random-number generation to develop a program that simulates the shuffling and dealing of cards. Once created, this program can be implemented in programs that imitate specific card games. We include several exercises at the end of this chapter that require card shuffling and dealing capabilities.

Class `Card` (Fig. 15.17) contains two `string` instance variables—`face` and `suit`—that store references to the face name and suit name of a specific card. The constructor for the class receives two `string`s that it uses to initialize `face` and `suit`. Method `ToString` (lines 20–24) creates a `string` consisting of the `face` of the card and the `suit` of the card.

We develop application `DeckForm` (Fig. 15.18), which creates a deck of 52 playing cards, using `Card` objects. Users can deal each card by clicking the **Deal Card** button. Each dealt card is displayed in a `Label`. Users can also shuffle the deck at any time by clicking the **Shuffle Cards** button.

Method `DeckForm_Load` (lines 35–53 of Fig. 15.18) uses the `for` structure (lines 50–51) to fill the `deck` array with `Card`s. Note that each `Card` is instantiated and initialized with two `string`s—one from the `faces` array (`string`s `"Ace"` through `"King"`) and one from the `suits` array (`"Hearts"`, `"Diamonds"`, `"Clubs"` or `"Spades"`). The calculation `i % 13` always results in a value from `0` to `12` (the thirteen subscripts of the `faces` array), and the calculation `i % 4` always results in a value from `0` to `3` (the four subscripts in the `suits` array). The initialized `deck` array contains the cards with faces ace through king for each suit.

```
1   // Fig. 15.17: Card.cs
2   // Stores suit and face information on each card.
3
4   using System;
5
6   // the representation of a card
7   public class Card
8   {
9      private string face;
10     private string suit;
11
12     public Card( string faceValue,
13        string suitValue )
14     {
15        face = faceValue;
16        suit = suitValue;
17
18     } // end constructor
19
```

Fig. 15.17 `Card` class. (Part 1 of 2.)

```
   public override string ToString()
   {
      return face + " of " + suit;

   } // end method ToString

} // end class Card
```

Fig. 15.17 `Card` class. (Part 2 of 2.)

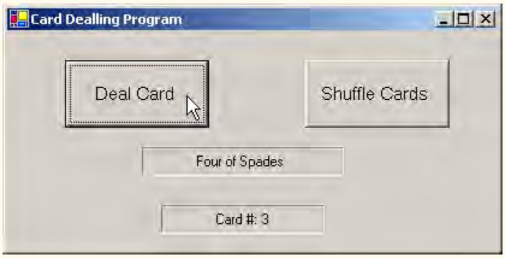
```
// Fig. 15.18: DeckOfCards.cs
// Simulating card drawing and shuffling.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// provides the functionality for the form
public class DeckForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button dealButton;
   private System.Windows.Forms.Button shuffleButton;

   private System.Windows.Forms.Label displayLabel;
   private System.Windows.Forms.Label statusLabel;

   private System.ComponentModel.Container components = null;

   private Card[] deck = new Card[ 52 ];
   private int currentCard;

   // main entry point for application
   [STAThread]
   static void Main()
   {
      Application.Run( new deckForm() );
   }

   // Visual Studio .NET generated code

   // handles form at load time
   private void DeckForm_Load(
      object sender, System.EventArgs e )
   {
      string[] faces = { "Ace", "Deuce", "Three", "Four",
                         "Five", "Six", "Seven", "Eight",
                         "Nine", "Ten", "Jack", "Queen",
                         "King" };
```

Fig. 15.18 Card dealing and shuffling simulation. (Part 1 of 4.)

```

      string[] suits = { "Hearts", "Diamonds", "Clubs",
                         "Spades" };

      // no cards have been drawn
      currentCard = -1;

      // initialize deck
      for ( int i = 0; i < deck.Length; i++ )
         deck[ i ] = new Card( faces[ i % 13 ], suits[ i % 4 ] );

   } // end method deckForm_Load

   // handles dealButton Click
   private void dealButton_Click(
      object sender, System.EventArgs e )
   {
      Card dealt = DealCard();

      // if dealt card is null, then no cards left
      // player must shuffle cards
      if ( dealt != null )
      {
         displayLabel.Text = dealt.ToString();
         statusLabel.Text = "Card #: " + currentCard;
      }
      else
      {
         displayLabel.Text = "NO MORE CARDS TO DEAL";
         statusLabel.Text = "Shuffle cards to continue";
      }
   }

   // shuffle cards
   private void Shuffle()
   {
      Random randomNumber = new Random();
      Card temporaryValue;

      currentCard = -1;

      // swap each card with random card
      for ( int i = 0; i < deck.Length; i++ )
      {
         int j = randomNumber.Next( 52 );

         // swap cards
         temporaryValue = deck[ i ];
         deck[ i ] = deck[ j ];
         deck[ j ] = temporaryValue;
      }

```

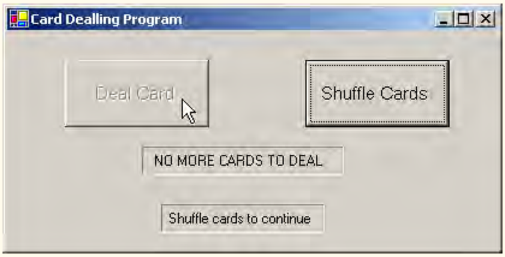
Fig. 15.18 Card dealing and shuffling simulation. (Part 2 of 4.)

```
      dealButton.Enabled = true;

   } // end method Shuffle

   private Card DealCard()
   {
      // if there is a card to deal then deal it
      // otherwise signal that cards need to be shuffled by
      // disabling dealButton and returning null
      if ( currentCard + 1 < deck.Length )
      {
         currentCard++;
         return deck[ currentCard ];
      }
      else
      {
         dealButton.Enabled = false;
         return null;
      }

   } // end method DealCard

   // handles shuffleButton Click
   private void shuffleButton_Click(
      object sender, System.EventArgs e )
   {
      displayLabel.Text = "SHUFFLING...";
      Shuffle();
      displayLabel.Text = "DECK IS SHUFFLED";

   } // end method shuffleButton_Click

} // end class deckForm
```


Card Dealling Program
Deal Card
Shuffle Cards

Four of Spades

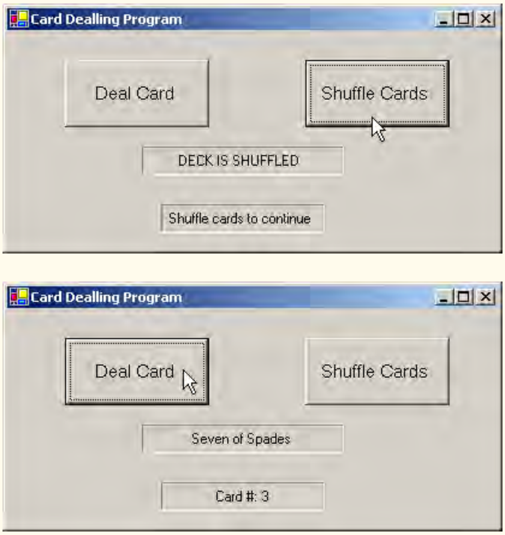
Card #: 3


Card Dealling Program
Deal Card
Shuffle Cards

NO MORE CARDS TO DEAL

Shuffle cards to continue

Fig. 15.18 Card dealing and shuffling simulation. (Part 3 of 4.)



DECK IS SHUFFLED

Shuffle cards to continue

Seven of Spades

Card #: 3

Fig. 15.18 Card dealing and shuffling simulation. (Part 4 of 4.)

When users click the **Deal Card** button, event handler `dealButton_Click` (lines 56–73) invokes method `DealCard` (defined in lines 98–114) to get the next card in the `deck` array. If the `deck` is not empty, the method returns a `Card` object reference; otherwise, it returns `null`. If the reference is not `null`, lines 65–66 display the `Card` in `displayLabel` and display the card number in the `statusLabel`.

If `DealCard` returns a `null` reference, the `string` `"NO MORE CARDS TO DEAL"` is displayed in `displayLabel`, and the `string` `"Shuffle cards to continue"` is displayed in `statusLabel`.

When users click the **Shuffle Cards** button, its event-handling method `shuffleButton_Click` (lines 117–124) invokes method `Shuffle` (defined on lines 76–96) to shuffle the cards. The method loops through all 52 cards (array subscripts `0`–`51`). For each card, the method randomly picks a number between `0` and `51`. Then the current `Card` object and the randomly selected `Card` object are swapped in the array. To shuffle the cards, method `Shuffle` makes a total of only 52 swaps during a single pass of the entire array. When the shuffling is complete, `displayLabel` displays the `string` `"DECK IS SHUFFLED"`.

15.17 Regular Expressions and Class `Regex`

Regular expressions are specially formatted `string`s used to find patterns in text and can be useful during information validation, to ensure that data is in a particular format. For example, a ZIP code must consist of five digits, and a last name must start with a capital letter. One application of regular expressions is to facilitate the construction of a compiler. Often, a large and complex regular expression is used to validate the syntax of a program. If the program code does not match the regular expression, the compiler knows that there is a syntax error within the code.

The .NET Framework provides several classes to help developers recognize and manipulate regular expressions. Class **`Regex`** (**`System.Text.RegularExpressions`** namespace) represents an immutable regular expression. It contains static methods that allow use of the **`Regex`** class without explicitly instantiating objects of that class. Class **`Match`** represents the results of a regular expression matching operation.

Class **`Regex`** provides method **`Match`**, which returns an object of class **`Match`** that represents a single regular expression match. **`Regex`** also provides method **`Matches`**, which finds all matches of a regular expression in an arbitrary **`string`** and returns a ***`MatchCollection`*** object—i.e., a set of **`Match`**es.

Common Programming Error 15.4

*When using regular expressions, do not confuse class **`Match`** with the method **`Match`**, which belongs to class **`Regex`**.*

Common Programming Error 15.5

*Visual Studio does not add **`System.Text.RegularExpressions`** to the list of namespaces imported in the project properties, so a programmer must import it manually with the statement **`using System.Text.RegularExpressions`**.*

The table in Fig. 15.19 specifies some *character classes* that can be used with regular expressions. A character class is an escape sequence that represents a group of characters.

A *word character* is any alphanumeric character or underscore. A *whitespace* character is a space, a tab, a carriage return, a newline or a form feed. A *digit* is any numeric character. Regular expressions are not limited to these character classes, however. The expressions employ various operators and other forms of notation to search for complex patterns. We discuss several of these techniques in the context of the next example.

Figure 15.20 presents a simple example that employs regular expressions. This program takes birthdays and tries to match them to a regular expression. The expression matches only birthdays that do not occur in April and that belong to people whose names begin with **`"J"`**.

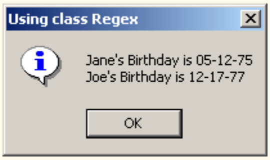
Character	Matches	Character	Matches
`\d`	any digit	`\D`	any non-digit
`\w`	any word character	`\W`	any non-word character
`\s`	any whitespace	`\S`	any non-whitespace

Fig. 15.19 Character classes.

```
// Fig. 15.20: RegexMatches.cs
// Demonstrating Class Regex.

using System;
using System.Windows.Forms;
using System.Text.RegularExpressions;

```

Fig. 15.20 Regular expressions checking birthdays. (Part 1 of 2.)

```
 8  // test out regular expressions
 9  class RegexMatches
10  {
11     // The main entry point for the application.
12     [STAThread]
13     static void Main( string[] args )
14     {
15        string output = "";
16
17        // create regular expression
18        Regex expression =
19           new Regex( @"J.*\d[0-35-9]-\d\d-\d\d" );
20
21        string string1 = "Jane's Birthday is 05-12-75\n" +
22           "Dave's Birthday is 11-04-68\n" +
23           "John's Birthday is 04-28-73\n" +
24           "Joe's Birthday is 12-17-77";
25
26        // match regular expression to string and
27        // print out all matches
28        foreach ( Match myMatch in expression.Matches( string1 ) )
29           output += myMatch.ToString() + "\n";
30
31        MessageBox.Show( output, "Using class Regex",
32           MessageBoxButtons.OK, MessageBoxIcon.Information );
33
34     } // end method Main
35
36  } // end class RegexMatches
```



```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Fig. 15.20 Regular expressions checking birthdays. (Part 2 of 2.)

Line 19 creates an instance of class `Regex` and defines the regular expression pattern for which `Regex` will search. The first character in the regular expression, `"J"`, is treated as a literal character. This means that any `string` matching this regular expression is required to start with `"J"`.

In a regular expression, the dot character `"."` matches any single character except a newline character. However, when the dot character is followed by an asterisk, as in the expression `".*"`, it matches any number of unspecified characters. In general, when the operator `"*"` is applied to any expression, the expression will match zero or more occurrences of the expression. By contrast, the application of the operator `"+"` to an expression causes the expression to match one or more occurrences of that expression. For example, both `"A*"` and `"A+"` will match `"A"`, but only `"A*"` will match an empty `string`.

As indicated in Fig. 15.19, `"\d"` matches any numeric digit. To specify sets of characters other than those that have a character class, characters can be listed in square brackets, `[]`. For example, the pattern `"[aeiou]"` can be used to match any vowel. Ranges of characters can be represented by placing a dash (`-`) between two characters. In the example, `"[0-35-9]"` matches only digits in the ranges specified by the pattern. In this case, the pattern matches any digit between `0` and `3` or between `5` and `9`; therefore, it matches any digit except `4`. If the first character in the brackets is the `"^"`, the expression accepts any character other than those indicated. However, it is important to note that `"[^4]"` is not the same as `"[0-35-9]"`; the former matches any nondigit, in addition to the digits other than `4`.

Although the `"-"` character indicates a range when it is enclosed in square brackets, instances of the `"-"` character outside grouping expressions are treated as literal characters. Thus, the regular expression in line 19 searches for a **`string`** that starts with the letter **`"J"`**, followed by any number of characters, followed by a two-digit number (of which the second digit cannot be `4`), followed by a dash, another two-digit number, a dash and another two-digit number.

Lines 28–29 use a **`foreach`** loop to iterate through each **`Match`** obtained from **`expression.Matches`**, which used **`string1`** as an argument. The output in Fig. 15.20 indicates the two matches that were found in **`string1`**. Notice that both matches conform to the pattern specified by the regular expression.

The asterisk (`*`) and plus (`+`) in the previous example are called *quantifiers*. Figure 15.21 lists various quantifiers and their uses.

We have already discussed how the asterisk (`*`) and plus (`+`) work. The question mark (`?`) matches zero or one occurrences of the expression that it quantifies. A set of braces containing one number (**`{n}`**) matches exactly **`n`** occurrences of the expression it quantifies. We demonstrate this quantifier in the next example. Including a comma after the number enclosed in braces matches at least **`n`** occurrences of the quantified expression. The set of braces containing two numbers (**`{n,m}`**), matches between **`n`** and **`m`** occurrences of the expression that it qualifies. All of the quantifiers are *greedy*. This means that they will match as many occurrences as they can as long as the match is successful. However, if any of these quantifiers is followed by a question mark (`?`), the quantifier becomes *lazy*. It then will match as few occurrences as possible as long as the match is successful.

The Windows application in Fig. 15.22 presents a more involved example that validates user input via regular expressions.

When a user clicks the **OK** button, the program checks to make sure that none of the fields is empty (lines 49–52). If one or more fields are empty, the program signals the user that all fields must be filled before the program can validate the input information (lines 55–56). Line 59 calls instance method **`Focus`** of class **`TextBox`**. Method **`Focus`** places the cursor within the **`TextBox`** that made the call. The program then exits the event handler (line 61). If there are no empty fields, the user input is validated. The **Last Name** is validated first (lines 65–74). If it passes the test (i.e., if the ***`Success`*** property of the **`Match`** instance is **`true`**), control moves on to validate the **First Name** (lines 77–86). This process continues until all **`TextBox`**es are validated or until a test fails (**`Success`** is **`false`**) and the program sends an appropriate error message. If all fields contain valid information, success is signaled, and the program quits.

In the previous example, we searched for substrings that matched a regular expression. In this example, we want to check whether an entire **`string`** conforms to a regular expression. For example, we want to accept **`"Smith"`** as a last name, but not **`"9@Smith#"`**. We achieve this effect by beginning each regular expression with a `"^"` character and ending it with a `"$"` character. The `"^"` and `"$"` characters match the positions at the beginning and end of a `string`, respectively. This forces the regular expression to evaluate the entire `string` and not return a match if a substring matches successfully.

In this program, we use the `static` version of `Regex` method ***`Match`***, which takes an additional parameter specifying the regular expression that we are trying to match. The expression in line 66 uses the square bracket and range notation to match an uppercase first letter, followed by letters of any case—`a-z` matches any lowercase letter, and `A-Z` matches any uppercase letter. The `*` quantifier signifies that the second range of characters may occur zero or more times in the `string`. Thus, this expression matches any `string` consisting of one uppercase letter, followed by zero or more additional letters.

The notation `\s` matches a single whitespace character (lines 90, 102 and 114). The expression `\d{5}`, used in the **Zip** (zip code) field, matches any five digits (line 125). In general, an expression with a positive integer `x` in the curly braces will match any `x` digits. (Notice the importance of the `"^"` and `"$"` characters to prevent zip codes with extra digits from being validated.)

Quantifier	Matches
`*`	Matches zero or more occurrences of the pattern.
`+`	Matches one or more occurrences of the pattern.
`?`	Matches zero or one occurrences of the pattern.
`{n}`	Matches exactly `n` occurrences.
`{n,}`	Matches at least `n` occurrences.
`{n,m}`	Matches between `n` and `m` (inclusive) occurrences.

Fig. 15.21 Quantifiers used regular expressions.

```
1   // Fig. 15.22: Validate.cs
2   // Validate user information using regular expressions.
3
4   using System;
5   using System.Drawing;
6   using System.Collections;
7   using System.ComponentModel;
8   using System.Windows.Forms;
9   using System.Data;
10  using System.Text.RegularExpressions;
11
12  // use regular expressions to validate strings
13  public class ValidateForm : System.Windows.Forms.Form
14  {
15     private System.Windows.Forms.Label phoneLabel;
16     private System.Windows.Forms.Label zipLabel;
17     private System.Windows.Forms.Label stateLabel;
18     private System.Windows.Forms.Label cityLabel;
19     private System.Windows.Forms.Label addressLabel;
```

Fig. 15.22 Validating user information using regular expressions. (Part 1 of 5.)

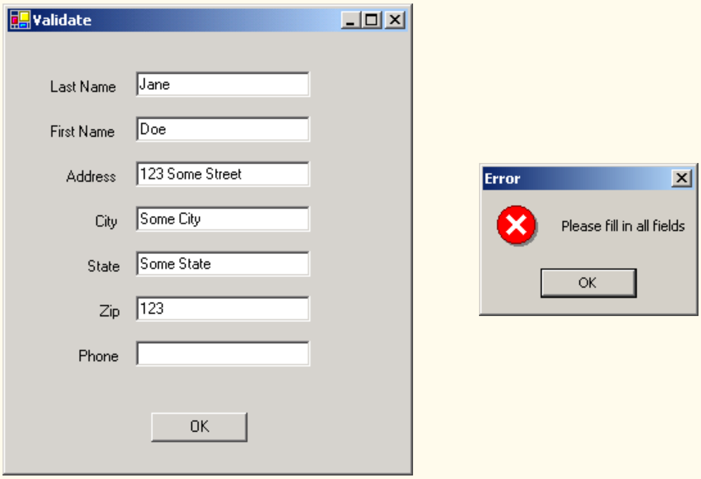
```
   private System.Windows.Forms.Label firstLabel;
   private System.Windows.Forms.Label lastLabel;

   private System.Windows.Forms.Button okButton;

   private System.Windows.Forms.TextBox phoneTextBox;
   private System.Windows.Forms.TextBox zipTextBox;
   private System.Windows.Forms.TextBox stateTextBox;
   private System.Windows.Forms.TextBox cityTextBox;
   private System.Windows.Forms.TextBox addressTextBox;
   private System.Windows.Forms.TextBox firstTextBox;
   private System.Windows.Forms.TextBox lastTextBox;

   private System.ComponentModel.Container components = null;

   // The main entry point for the application.
   [STAThread]
   static void Main()
   {
      Application.Run( new validateForm() );
   }

   // Visual Studio .NET generated code

   // handles okButton Click event
   private void okButton_Click(
      object sender, System.EventArgs e )
   {
      // ensures no textboxes are empty
      if ( lastTextBox.Text == "" || firstTextBox.Text == "" ||
         addressTextBox.Text == "" || cityTextBox.Text == "" ||
         stateTextBox.Text == "" || zipTextBox.Text == "" ||
         phoneTextBox.Text == "" )
      {
         // display popup box
         MessageBox.Show( "Please fill in all fields", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );

         // set focus to lastTextBox
         lastTextBox.Focus();

         return;
      }

      // if last name format invalid show message
      if ( !Regex.Match( lastTextBox.Text,
         @"^[A-Z][a-zA-Z]*$" ).Success )
      {
         // last name was incorrect
         MessageBox.Show( "Invalid Last Name", "Message",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
         lastTextBox.Focus();

```

Fig. 15.22 Validating user information using regular expressions. (Part 2 of 5.)

```
            return;
         }

         // if first name format invalid show message
         if ( !Regex.Match( firstTextBox.Text,
            @"^[A-Z][a-zA-Z]*$" ).Success )
         {
            // first name was incorrect
            MessageBox.Show( "Invalid First Name", "Message",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
            firstTextBox.Focus();

            return;
         }

         // if address format invalid show message
         if ( !Regex.Match( addressTextBox.Text,
           @"^[0-9]+\s+([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$" ).Success )
         {
            // address was incorrect
            MessageBox.Show( "Invalid Address", "Message",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
            addressTextBox.Focus();

            return;
         }

         // if city format invalid show message
         if ( !Regex.Match( cityTextBox.Text,
            @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$" ).Success )
         {
            // city was incorrect
            MessageBox.Show( "Invalid City", "Message",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
            cityTextBox.Focus();

            return;
         }

         // if state format invalid show message
         if ( !Regex.Match( stateTextBox.Text,
            @"^([a-zA-Z]+|[a-zA-Z]+\s[a-zA-Z]+)$" ).Success )
         {
            // state was incorrect
            MessageBox.Show( "Invalid State", "Message",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
            stateTextBox.Focus();

            return;
         }

```

Fig. 15.22 Validating user information using regular expressions. (Part 3 of 5.)

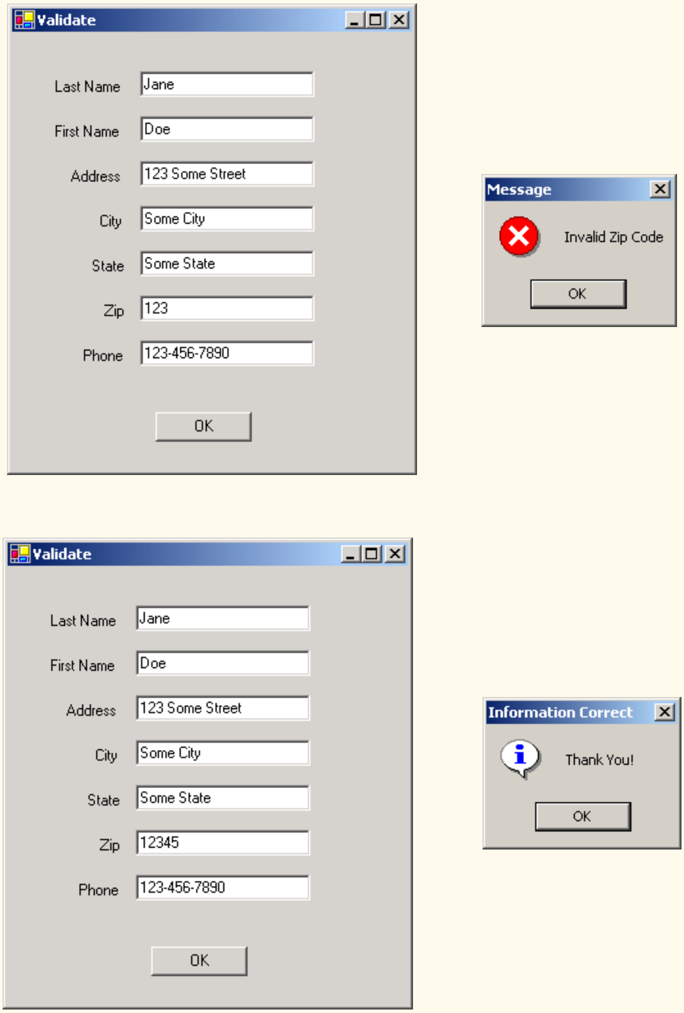
```
      // if zip code format invalid show message
      if ( !Regex.Match( zipTextBox.Text, @"^\d{5}$" ).Success )
      {
         // zip was incorrect
         MessageBox.Show( "Invalid Zip Code", "Message",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
         zipTextBox.Focus();

         return;
      }

      // if phone number format invalid show message
      if ( !Regex.Match( phoneTextBox.Text,
         @"^[1-9]\d{2}-[1-9]\d{2}-\d{4}$" ).Success )
      {
         // phone number was incorrect
         MessageBox.Show( "Invalid Phone Number", "Message",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
         phoneTextBox.Focus();

         return;
      }

      // information is valid, signal user and exit application
      this.Hide();
      MessageBox.Show( "Thank You!", "Information Correct",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

      Application.Exit();

   } // end method okButton_Click

} // end class ValidateForm
```



Last Name: Jane
First Name: Doe
Address: 123 Some Street
City: Some City
State: Some State
Zip: 123
Phone:

Please fill in all fields

Fig. 15.22 Validating user information using regular expressions. (Part 4 of 5.)



Validate

Last Name: Jane
First Name: Doe
Address: 123 Some Street
City: Some City
State: Some State
Zip: 123
Phone: 123-456-7890

Message

Invalid Zip Code

Validate

Last Name: Jane
First Name: Doe
Address: 123 Some Street
City: Some City
State: Some State
Zip: 12345
Phone: 123-456-7890

Information Correct

Thank You!

Fig. 15.22 Validating user information using regular expressions. (Part 5 of 5.)

The character "|" matches the expression to its left or to its right. For example, **`Hi (John|Jane)`** matches both **`Hi John`** and **`Hi Jane`**. Note the use of parentheses to group parts of the regular expression. Quantifiers may be applied to patterns enclosed in parentheses to create more complex regular expressions.

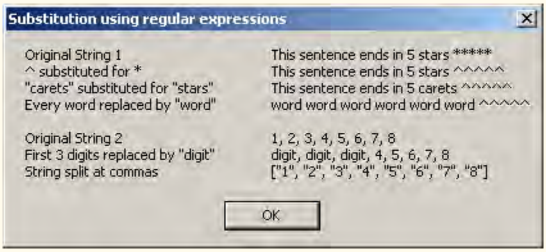
The **Last Name** and **First Name** fields both accept **`string`**s of any length, that begin with an uppercase letter. The **Address** field matches a number of at least one digit, followed by a space and then either one or more letters or else one or more letters followed by a space and another series of one or more letters (line 90). Therefore, **`"10 Broadway"`** and **`"10 Main Street"`** are both valid addresses. The **City** (line 102) and **State** (line 114) fields match any word of at least one character or, alternatively, any two words of at least one character if the words are separated by a single space. This means both **`Waltham`** and **`West Newton`** would match. As previously stated, the **Zip** code must be a five-digit number (line 125). The **Phone** number must be of the form **`xxx-yyy-yyyy`**, where the **`x`**s represent the area code and **`y`**s the number (line 137). The first **`x`** and the first **`y`** cannot be zero.

Sometimes it is useful to replace parts of a **`string`** with another, or split a **`string`** according to a regular expression. For this purpose, the **`Regex`** class provides **`static`** and instance versions of methods ***`Replace`*** and ***`Split`***, which are demonstrated in Fig. 15.23.

```
// Fig. 15.23: RegexSubstitution.cs
// Using Regex method Replace.

using System;
using System.Text.RegularExpressions;
using System.Windows.Forms;

// Summary description for RegexSubstitution.
public class RegexSubstitution1
{

   // The main entry point for the application.
   static void Main( string[] args )
   {
      string testString1 =
         "This sentence ends in 5 stars *****";

      string testString2 = "1, 2, 3, 4, 5, 6, 7, 8";
      Regex testRegex1 = new Regex( "stars" );
      Regex testRegex2 = new Regex( @"\d" );
      string[] results;
      string output = "Original String 1\t\t\t" + testString1;

      testString1 = Regex.Replace( testString1, @"\*", "^" );

      output += "\n^ substituted for *\t\t\t" + testString1;

      testString1 = testRegex1.Replace( testString1, "carets" );

      output += "\n\"carets\" substituted for \"stars\"\t" +
         testString1;

      output += "\nEvery word replaced by \"word\"\t" +
         Regex.Replace( testString1, @"\w+", "word" );

      output += "\n\nOriginal String 2\t\t\t" + testString2;

      output += "\nFirst 3 digits replaced by \"digit\"\t" +
         testRegex2.Replace( testString2, "digit", 3 );

      output += "\nString split at commas\t\t[";

      results = Regex.Split( testString2, @",\s*" );

```

Fig. 15.23 **`Regex`** methods **`Replace`** and **`Split`**. (Part 1 of 2.)

```
45          foreach ( string resultString in results )
46          {
47             output += "\"" + resultString + "\", ";
48          }
49
50          output = output.Substring( 0, output.Length - 2 ) + "]";
51
52          MessageBox.Show( output,
53             "Substitution using regular expressions" );
54
55       } // end method Main
56
57    } // end class RegexSubstitution
```



```
Original String 1                  This sentence ends in 5 stars *****
^ substituted for *                This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars"   This sentence ends in 5 carets ^^^^^
Every word replaced by "word"      word word word word word word ^^^^^

Original String 2                  1, 2, 3, 4, 5, 6, 7, 8
First 3 digits replaced by "digit" digit, digit, digit, 4, 5, 6, 7, 8
String split at commas             ["1", "2", "3", "4", "5", "6", "7", "8"]
```

Fig. 15.23 `Regex` methods `Replace` and `Split`. (Part 2 of 2.)

Method `Replace` replaces text in a `string` with new text wherever the original `string` matches a regular expression. We present two versions of this method in Fig. 15.23. The first version (line 24) is `static` and takes three parameters—the `string` to modify, the `string` containing the regular expression to match and the replacement `string`. Here, `Replace` replaces every instance of `"*"` in `testString1` with `"^"`. Notice that the regular expression (`@"\*"`) precedes character `*` with a backslash, `\`. Normally, `*` is a quantifier indicating that a regular expression should match any number of occurrences of a preceding pattern. However, in line 24, we want to find all occurrences of the literal character `*`; to do this, we must escape character `*` with character `\`. By escaping a special regular expression character with a `\`, we inform the regular-expression matching engine to find the actual character, as opposed to what it represents in a regular expression. The second version of method `Replace` (line 28) is an instance method that uses the regular expression passed to the constructor for `testRegex1` (line 19) to perform the replacement operation. In this case, every match for the regular expression `"stars"` in `testString1` is replaced with `"carets"`.

Line 20 instantiates `testRegex2` with argument `@"\d"`. The call to instance method `Replace` in line 39 takes three arguments—a `string` to modify, a `string` containing the replacement text and an `int` specifying the number of replacements to make. In other words, this version of `Replace` replaces the first three instances of a digit (`"\d"`) in `testString2` with the text `"digit"` (line 39).

Method `Split` divides a `string` into several substrings. The original `string` is broken in any location that matches a specified regular expression. Method `Split` returns an array containing the substrings between matches for the regular expression. In line 43, we use the `static` version of method `Split` to separate a `string` of comma-separated integers. The first argument is the **`string`** to split; the second argument is the regular expression. In this case, we use the regular expression **`@",\s*"`** to separate the substrings wherever a comma occurs. By matching any whitespace characters, we eliminate extra spaces from the resulting substrings.

SUMMARY

- Characters are the fundamental building blocks of C# program code. Every program is composed of a sequence of characters that is interpreted by the compiler as a series of instructions used to accomplish a task.
- A **`string`** is a series of characters treated as a single unit. A **`string`** may include letters, digits and various special characters, such as `+`, `-`, `*`, `/`, `$` and others.
- All characters correspond to numeric codes. When the computer compares two **`string`**s, it actually compares the numeric codes of the characters in the **`string`**s.
- Method **`Equals`** uses a lexicographical comparison, meaning that if a certain **`string`** has a higher value than another **`string`**, it would be found later in a dictionary. Method **`Equals`** compares the integer Unicode values that represent each character in each **`string`**.
- Method **`CompareTo`** returns `0` if the **`string`**s are equal, a negative number if the **`string`** that invokes **`CompareTo`** is less than the **`string`** passed as an argument, a positive number if the **`string`** that invokes **`CompareTo`** is greater than the **`string`** passed as an argument. Method **`CompareTo`** uses a lexicographical comparison.
- A hash table stores information, using a special calculation on the object to be stored that produces a hash code. The hash code is used to choose the location in the table at which to store the object.
- Class **`Object`** defines method **`GetHashCode`** to perform the hash-code calculation. This method is inherited by all subclasses of **`Object`**. Method **`GetHashCode`** is overridden by **`String`** to provide a good hash-code distribution based on the contents of the **`string`**.
- Class **`String`** provides two **`Substring`** methods to enable a new **`string`** to be created by copying part of an existing **`string`**.
- **`String`** method **`IndexOf`** locates the first occurrence of a character or a substring in a **`string`**. Method **`LastIndexOf`** locates the last occurrence of a character or a substring in a **`string`**.
- **`String`** method **`StartsWith`** determines whether a **`string`** starts with the characters specified as an argument. **`String`** method **`EndsWith`** determines whether a **`string`** ends with the characters specified as an argument.
- The **`static`** method **`Concat`** of class **`String`** concatenates two **`string`**s and returns a new **`string`** containing the characters from both original **`string`**s.
- Methods **`Replace`**, **`ToUpper`**, **`ToLower`**, **`Trim`** and **`Remove`** are provided for more advanced **`string`** manipulation.
- The **`String`** class provides many capabilities for processing **`string`**s. However, once a **`string`** is created, its contents can never change. Class **`StringBuilder`** is available for creating and manipulating dynamic **`string`**s, i.e., **`string`**s that can change.
- Class **`StringBuilder`** provides **`Length`** and **`Capacity`** properties to return the number of characters currently in a **`StringBuilder`** and the number of characters that can be stored in a **`StringBuilder`** without allocating more memory, respectively. These properties also can be used to increase or decrease the length or the capacity of the **`StringBuilder`**.

- Method **`EnsureCapacity`** allows programmers to guarantee that a **`StringBuilder`** has a minimum capacity. Method **`EnsureCapacity`** attempts to double the capacity. If this value is greater than the value that the programmer wishes to ensure, this will be the new capacity. Otherwise, **`EnsureCapacity`** alters the capacity to make it one more than the requested number.
- Class **`StringBuilder`** provides 19 overloaded **`Append`** methods to allow various data-type values to be added to the end of a **`StringBuilder`**. Versions are provided for each of the primitive data types and for character arrays, **`String`**s and **`Object`**s.
- The braces in a format **`string`** specify how to format a specific piece of information. Formats have the form **`{X[,Y][:FormatString]}`**, where **`X`** is the number of the argument to be formatted, counting from zero. **`Y`** is an optional argument, which can be positive or negative. **`Y`** indicates how many characters should be in the result of formatting; if the resulting **`string`** is less than this number, it will be padded with spaces to make up for the difference. A positive integer means the **`string`** will be right aligned; a negative one means it will be left aligned. The optional **`FormatString`** indicates what kind of formatting should be applied to the argument: Currency, decimal, or scientific, among others.
- Class **`StringBuilder`** provides 19 overloaded **`Insert`** methods to allow various data-type values to be inserted at any position in a **`StringBuilder`**. Versions are provided for each of the primitive data types and for character arrays, **`String`**s and **`Object`**s.
- Class **`StringBuilder`** also provides method **`Remove`** for deleting any portion of a **`StringBuilder`**.
- Another useful method included with **`StringBuilder`** is **`Replace`**. **`Replace`** searches for a specified **`string`** or character and substitutes another in its place.
- C# provides **`struct`**s, program building blocks similar to classes.
- Structures are in many ways similar to classes, the largest difference between them being that structures encapsulate value types, whereas classes encapsulate reference types.
- Many of the primitive data types that we have been using are actually aliases for different structures. These structures are derived from class **`ValueType`**, which in turn is derived from class **`Object`**.
- **`Char`** is a structure that represents characters.
- Method **`Char.Parse`** converts data into a character.
- Method **`Char.IsDigit`** determines whether a character is a defined Unicode digit.
- Method **`Char.IsLetter`** determines whether a character is a letter.
- Method **`Char.IsLetterOrDigit`** determines whether a character is a letter or a digit.
- Method **`Char.IsLower`** determines whether a character is a lowercase letter.
- Method **`Char.IsUpper`** determines whether a character is an uppercase letter.
- Method **`Char.ToUpper`** converts a character to its uppercase equivalent.
- Method **`Char.ToLower`** converts a character to its lowercase equivalent.
- Method **`Char.IsPunctuation`** determines whether a character is a punctuation mark.
- Method **`Char.IsSymbol`** determines whether a character is a symbol.
- Method **`Char.IsWhiteSpace`** determines whether a character is a whitespace character.
- **`Char`** method **`CompareTo`** compares two character values.
- Regular expressions find patterns in text.
- The .NET Framework provides class **`Regex`** to aid developers in recognizing and manipulating regular expressions. **`Regex`** provides method **`Match`**, which returns an object of class **`Match`**. This object represents a single match in a regular expression. **`Regex`** also provides the method

`Matches`, which finds all matches of a regular expression in an arbitrary `string` and returns a `MatchCollection`—a set of `Match`es.

- Both classes `Regex` and `Match` are in namespace `System.Text.RegularExpressions`.
- In general, applying the quantifier `*` to any expression will match zero or more occurrences of that expression, and applying the quantifier `+` will match one or more occurrences of that expression.
- The pattern `"[0-35-9]"` is a regular expression that matches one in a range of characters. This `string` will match any digit `0-3` and `5-9`, so it will match any digit except `4`.
- The character "`|`" matches the expression to its left or to its right. For example, `"Hi (John|Jane)"` matches both `"Hi John"` and `"Hi Jane"`.
- Method `Replace` replaces those substrings in a `string` that match a certain regular expression with a specified `string`.

TERMINOLOGY

`+` operator
`+=` concatenation operator
`==` comparison operator
alphabetizing
`Append` method of class `StringBuilder`
`AppendFormat` method of class `StringBuilder`
`ArgumentOutOfRangeException`
`Capacity` property of class `StringBuilder`
`char` array
`Char` structure
`Chars` property of class `String`
character
character class
`CompareTo` method of class `String`
`CompareTo` method of structure `Char`
`Concat` method of class `String`
`CopyTo` method of class `String`
`Enabled` property of class `Control`
`EndsWith` method of class `String`
`EnsureCapacity` method of class `StringBuilder`
`Equals` method of class `String`
format string
garbage collector
`GetHashCode`
greedy quantifier
hash code
hash table
immutable `String`
`IndexOf` method of class `String`
`IndexOfAny` method of class `String`
`IsDigit` method of structure `Char`
`IsLetter` method of structure `Char`
`IsLetterOrDigit` method of structure `Char`
`IsLower` method of structure `Char`
`IsPunctuation` method of structure `Char`
`IsSymbol` method of structure `Char`
`IsUpper` method of structure `Char`
`IsWhiteSpace` method of structure `Char`
`LastIndexOf` method of class `String`
`LastIndexOfAny` method of class `String`
lazy quantifier
`Length` property of class `String`
`Length` property of class `StringBuilder`
lexicographical comparison
literal strings
`Match` class
`MatchCollection` class
page-layout software
`Parse` method of structure `Char`
quantifier
random-number generation
`Regex` class
`Remove` method of class `StringBuilder`
`Replace` method of class `Regex`
`Replace` method of class `String`
`Replace` method of class `StringBuilder`
special characters
`Split` method of class `Regex`
`StartsWith` method of class `String`
`String` class
string literal
`string` reference
`StringBuilder` class
`struct`
`Substring` method of class `String`
`Success` property of class `Match`

`System` namespace
`System.Text` namespace
`System.Text.RegularExpressions` namespace
text editor
`ToLower` method of class `String`
`ToLower` method of structure `Char`
`ToString` method of class `String`
`ToString` method of class `StringBuilder`
`ToUpper` method of class `String`
`ToUpper` method of structure `Char`
trailing whitespace characters
`Trim` method of class `String`
Unicode character set
`ValueType` class
whitespace characters
word character

SELF-REVIEW EXERCISES

15.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) When `string`s are compared with `==`, the result is *true* if the `string`s contain the same values.
b) A `string` can be modified after it is created.
c) Class `String` has no `ToString` method.
d) `StringBuilder` method `EnsureCapacity` sets the `StringBuilder` instance's capacity to the argument's value.
e) The method `Equals` and the equality operator work the same for `string`s.
f) Method `Trim` removes all whitespace at the beginning and the end of a `string`.
g) A regular expression matches a `string` to a pattern.
h) It is always better to use `string`s rather than `StringBuilder`s because `string`s containing the same value will reference the same object in memory.
i) Class `String` method `ToUpper` capitalizes just the first letter of the `string`.
j) The expression `\d` in a regular expression denotes all letters.

15.2 Fill in the blanks in each of the following statements:

a) To concatenate `string`s, use the __________ operator or class __________ method __________.
b) Method `Compare` of class `String` uses a __________ comparison of `string`s.
c) Class `Regex` is located in namespace __________.
d) `StringBuilder` method __________ first formats the specified `string`, then concatenates it to the end of the `StringBuilder`.
e) If the arguments to a `Substring` method call are out of range, an __________ exception is thrown.
f) `Regex` method __________ changes all occurrences of a pattern in a `string` to a specified `string`.
g) Method __________ is inherited by every object and calculates its hash code.
h) A `C` in a format string means to output the number as __________.
i) Regular expression quantifier __________ matches zero or more occurrences of an expression.
j) Regular expression operator __________ inside square brackets will not match any of the characters in that set of brackets.

ANSWERS TO SELF-REVIEW EXERCISES

15.1 a) True. b) False. `string`s are immutable and cannot be modified after they are created. `StringBuilder` objects can be modified after they are created. c) False. Class `String` inherits a `ToString` method from class `Object`. d) True. e) True. f) True. g) True. h) False. `StringBuilder` should be used if the `string` is to be modified. i) False. Class `String` method `ToUpper` capitalizes all letters in the `string`. j) False. The expression `\d` denotes all decimals in a regular expression.

15.2 a) **+**, **`StringBuilder`**, **`Append`**. b) lexicographical. c) **`System.Text.RegularExpressions`**. d) **`AppendFormat`** e) **`ArgumentOutOfRangeException`**. f) **`Replace`**. g) **`GetHashCode`**. h) currency. i) **`*`**. j) **`^`**.

EXERCISES

15.3 Modify the program in Fig. 15.18 so that the card-dealing method deals a five-card poker hand. Then write the following additional methods:

- a) Determine if the hand contains a pair.
- b) Determine if the hand contains two pairs.
- c) Determine if the hand contains three of a kind (e.g., three jacks).
- d) Determine if the hand contains four of a kind (e.g., four aces).
- e) Determine if the hand contains a flush (i.e., all five cards of the same suit).
- f) Determine if the hand contains a straight (i.e., five cards of consecutive face values).
- g) Determine if the hand contains a full house (i.e., two cards of one face value and three cards of another face value).

15.4 Use the methods developed in Exercise 15.3 to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

15.5 Write an application that uses **`String`** method **`CompareTo`** to compare two **`string`**s input by the user. Output whether the first **`string`** is less than, equal to or greater than the second.

15.6 Write an application that uses random-number generation to create sentences. Use four arrays of **`string`**s, called **`article`**, **`noun`**, **`verb`** and **`preposition`**. Create a sentence by selecting a word at random from each array in the following order: **`article`**, **`noun`**, **`verb`**, **`preposition`**, **`article`** and **`noun`**. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 sentences and output them to a text area.

The arrays should be filled as follows: The **`article`** array should contain the articles **`"the"`**, **`"a"`**, **`"one"`**, **`"some"`** and **`"any"`**; the **`noun`** array should contain the nouns **`"boy"`**, **`"girl"`**, **`"dog"`**, **`"town"`** and **`"car"`**; the **`verb`** array should contain the past-tense verbs **`"drove"`**, **`"jumped"`**, **`"ran"`**, **`"walked"`** and **`"skipped"`**; the **`preposition`** array should contain the prepositions **`"to"`**, **`"from"`**, **`"over"`**, **`"under"`** and **`"on"`**.

After the preceding program is written, modify the program to produce a short story consisting of several of these sentences. (How about the possibility of a random term-paper writer!)

15.7 (*Pig Latin*) Write an application that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm:

To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters “**`ay`**.” Thus, the word “**`jump`**” becomes “**`umpjay`**,” the word “**`the`**” becomes “**`hetay`**” and the word “**`computer`**” becomes “**`omputercay`**.” Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Enable the user to input a sentence. Use techniques discussed in this chapter to divide the sentence into separate words. Method **`GetPigLatin`** should translate a single word into pig Latin. Keep a running display of all the converted sentences in a text area.

15.8 Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word “bathe” include the commonly used words “ate,” “bat,” “bet,” “tab,” “hat,” “the” and “tea.”

16

Graphics and Multimedia

Objectives

- To understand graphics contexts and graphics objects.
- To be able to manipulate colors and fonts.
- To understand and be able to use GDI+ **`Graphics`** methods to draw lines, rectangles, **`string`**s and images.
- To be able to use class **`Image`** to manipulate and display images.
- To be able to draw complex shapes from simple shapes with class **`GraphicsPath`**.
- To be able to use Windows Media Player and Microsoft Agent in a C# application.

One picture is worth ten thousand words.
Chinese proverb

Treat nature in terms of the cylinder, the sphere, the cone, all in perspective.
Paul Cezanne

Nothing ever becomes real till it is experienced—even a proverb is no proverb to you till your life has illustrated it.
John Keats

A picture shows me at a glance what it takes dozens of pages of a book to expound.
Ivan Sergeyevich

Outline

16.1 Introduction
16.2 Graphics Contexts and Graphics Objects
16.3 Color Control
16.4 Font Control
16.5 Drawing Lines, Rectangles and Ovals
16.6 Drawing Arcs
16.7 Drawing Polygons and Polyli]nes
16.8 Advanced Graphics Capabilities
16.9 Introduction to Multimedia
16.10 Loading, Displaying and Scaling Images
16.11 Animating a Series of Images
16.12 Windows Media Player
16.13 Microsoft Agent

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

16.1 Introduction

In this chapter, we overview C#'s tools for drawing two-dimensional shapes and for controlling colors and fonts. C# supports graphics that enable programmers to enhance their Windows applications visually. The FCL contains many sophisticated drawing capabilities as part of namespace ***`System.Drawing`*** and the other namespaces that make up the .NET resource *GDI+*. GDI+, an extension of the Graphical Device Interface, is an application programming interface (API) that provides classes for creating two-dimensional vector graphics (a way of describing graphics so that they may be easily manipulated with high-performance techniques), manipulating fonts and inserting images. GDI+ expands GDI by simplifying the programming model and introducing several new features, such as graphics paths, extended image file format support and alpha blending. Using the GDI+ API, programmers can create images without worrying about the platform-specific details of their graphics hardware.

We begin with an introduction to the .NET framework's drawing capabilities. We then present more powerful drawing capabilities, such as changing the styles of lines used to draw shapes and controlling the colors and patterns of filled shapes.

Figure 16.1 depicts a portion of the `System.Drawing` class hierarchy, which includes several of the basic graphics classes and structures covered in this chapter. The most commonly used components of GDI+ reside in the `System.Drawing` and ***`System.Drawing.Drawing2D`*** namespaces.

Class ***`Graphics`*** contains methods used for drawing `string`s, lines, rectangles and other shapes on a `Control`. The drawing methods of class `Graphics` usually require a ***`Pen`*** or ***`Brush`*** object to render a specified shape. The `Pen` draws shape outlines; the `Brush` draws solid objects.

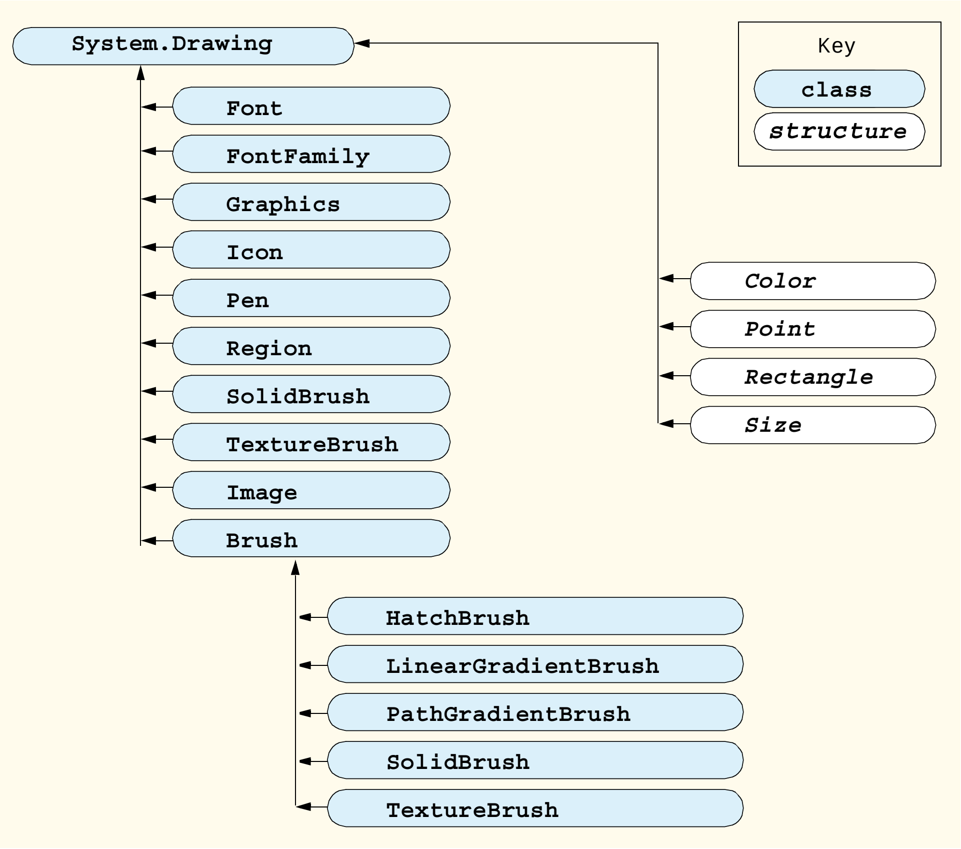

System.Drawing
Key
class
structure
Font
FontFamily
Graphics
Icon
Pen
Region
SolidBrush
TextureBrush
Image
Brush
Color
Point
Rectangle
Size
HatchBrush
LinearGradientBrush
PathGradientBrush
SolidBrush
TextureBrush

Fig. 16.1 **`System.Drawing`** namespace's classes and structures.

Structure **`Color`** contains numerous **`static`** properties, which set the colors of various graphical components, plus methods that allow users to create new colors. Class ***`Font`*** contains properties that define unique fonts. Class ***`FontFamily`*** contains methods for obtaining font information.

To begin drawing in C#, we first must understand GDI+'s *coordinate system* (Fig. 16.2), a scheme for identifying every point on the screen. By default, the upper-left corner of a GUI component (such as a **`Panel`** or a **`Form`**) has the coordinates (0, 0). A coordinate pair has both an *x-coordinate* (the *horizontal coordinate*) and a *y-coordinate* (the *vertical coordinate*). The *x*-coordinate is the horizontal distance (to the right) from the upper-left corner. The *y*-coordinate is the vertical distance (downward) from the upper-left corner. The *x-axis* defines every horizontal coordinate, and the *y-axis* defines every vertical coordinate. Programmers position text and shapes on the screen by specifying their (*x*,*y*) coordinates. Coordinate units are measured in *pixels* ("picture elements"), which are the smallest units of resolution on a display monitor.

The **`System.Drawing`** namespace provides structures **`Rectangle`** and **`Point`**. The ***`Rectangle`*** *structure* defines rectangular shapes and dimensions. The ***`Point`*** *structure* represents the *x*-*y* coordinates of a point on a two-dimensional plane.

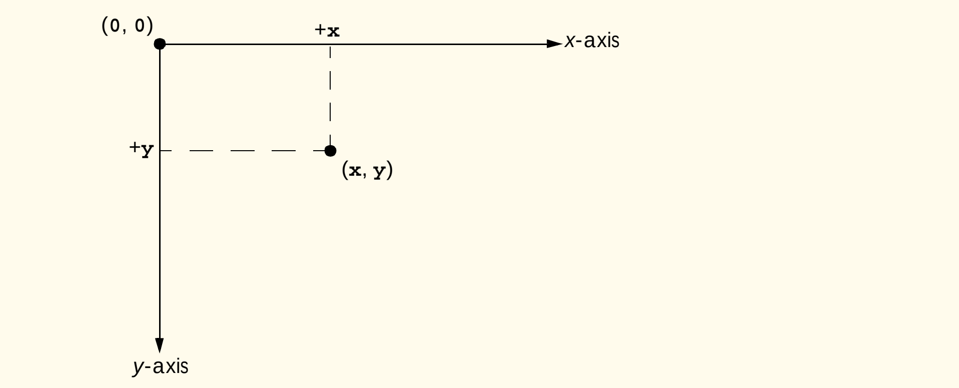


Fig. 16.2 GDI+ coordinate system. Units are measured in pixels.

Portability Tip 16.1

Different display monitors have different resolutions, so the density of pixels on such monitors will vary. This might cause the sizes of graphics to appear different on different monitors.

In the remainder of this chapter, we explore techniques for manipulating images and creating smooth animations. We also discuss class ***`Image`***, which can store and manipulate images from various file formats. Later, we explain how to combine the graphical rendering capabilities covered in the early sections of the chapter with those for image manipulation.

16.2 Graphics Contexts and Graphics Objects

A C# *graphics context* represents a drawing surface that enables drawing on the screen. A **`Graphics`** object manages a graphics context by controlling how information is drawn. **`Graphics`** objects contain methods for drawing, font manipulation, color manipulation and other graphics-related actions. Every Windows application that derives from class **`System.Windows.Forms.Form`** inherits an **`virtual`** ***`OnPaint`*** event handler where most graphics operations are performed. The arguments to the **`OnPaint`** method include a **`PaintEventArgs`** object from which we can obtain a **`Graphics`** object for the control. We must obtain the **`Graphics`** object on each call to the method, because the properties of the graphics context that the graphics object represents could change. The **`OnPaint`** method triggers the **`Control`**’s ***`Paint`*** event.

When displaying graphical information on a **`Form`**’s client area, programmers can override the **`OnPaint`** method to retrieve a **`Graphics`** object from argument **`PaintEventArgs`** or to create a new **`Graphics`** object associated with the appropriate surface. We demonstrate these techniques of drawing in C# later in the chapter.

To override the inherited **`OnPaint`** method, use the following method definition:

```
protected override void OnPaint( PaintEventArgs e )
```

Next, extract the incoming **`Graphics`** object from the **`PaintEventArgs`** argument:

```
Graphics graphicsObject = e.Graphics;
```

Variable **`graphicsObject`** now is available to draw shapes and **`string`**s on the form.

Calling the **`OnPaint`** method raises the **`Paint`** event. Instead of overriding the **`OnPaint`** method, programmers can add an event handler for the **`Paint`** event. Visual Studio .NET generates the **`Paint`** event handler in this form:

```
protected void MyEventHandler_Paint(
   object sender, PaintEventArgs e )
```

Programmers seldom call the **`OnPaint`** method directly, because the drawing of graphics is an *event-driven process*. An event—such as the covering, uncovering or resizing of a window—calls the **`OnPaint`** method of that form. Similarly, when any control (such as a **`TextBox`** or **`Label`**) is displayed, the program calls that control's **`Paint`** method.

If programmers need to cause method **`OnPaint`** to run explicitly, they should not call method **`OnPaint`**. Rather, they can call the ***`Invalidate`*** method (inherited from **`Control`**). This method refreshes a control's client area and implicitly repaints all graphical components. C# contains several overloaded **`Invalidate`** methods that allow programmers to update portions of the client area.

Performance Tip 16.1

*Calling the **`Invalidate`** method to refresh the **`Control`** often is inefficient. Instead, call **`Invalidate`** with a **`Rectangle`** parameter to refresh only the area designated by the rectangle. This improves program performance.*

Controls, such as **`Label`**s and **`Button`**s, do not have their own graphics contexts, but one can be created. To draw on a control, first create its graphics object by invoking the **`CreateGraphics`** method:

```
Graphics graphicsObject = controlName.CreateGraphics();
```

where *graphicsObject* represents an instance of class **`Graphics`** and *controlName* is any control. Now, a programmer can use the methods provided in class **`Graphics`** to draw on the control.

16.3 Color Control

Colors can enhance a program's appearance and help convey meaning. For example, a red traffic light indicates stop, yellow indicates caution and green indicates go.

Structure **`Color`** defines methods and constants used to manipulate colors. Because it is a lightweight object that performs only a handful of operations and stores **`static`** fields, **`Color`** is implemented as a structure, rather than as a class.

Every color can be created from a combination of alpha, red, green and blue components. Together, these components are called *ARGB values*. All four ARGB components are **`byte`**s that represent integer values in the range from 0 to 255. The alpha value determines the opacity of the color. For example, the alpha value 0 results in a transparent color, the value 255 in an opaque color. Alpha values between 0 and 255 result in a weighted blending effect of the color's RGB value with that of any background color, causing a semitransparent effect. The first number in the RGB value defines the amount of red in the color, the second defines the amount of green and the third defines the amount of blue. The larger the value, the greater the amount of that particular color. C# enables programmers to choose from almost 17 million colors. If a particular computer cannot display all these colors, it will display the color closest to the one specified. Figure 16.3 summarizes some predefined **`Color`** constants, and Fig. 16.4 describes several **`Color`** methods and properties.

The table in Fig. 16.4 describes two ***`FromArgb`*** method calls. One takes three **`int`** arguments, and one takes four **`int`** arguments (all argument values must be between 0 and 255). Both take **`int`** arguments specifying the amount of red, green and blue. The overloaded version takes four arguments and allows the user to specify alpha; the three-argument version defaults the alpha to 255. Both methods return a **`Color`** object representing the specified values. **`Color`** properties ***`A`***, ***`R`***, ***`G`*** and ***`B`*** return **`byte`**s that represent **`int`** values from 0 to 255, corresponding to the amounts of alpha, red, green and blue, respectively.

Programmers draw shapes and **`string`**s with **`Brush`**es and **`Pen`**s. A **`Pen`**, which functions similarly to an ordinary pen, is used to draw lines. Most drawing methods require a **`Pen`** object. The overloaded **`Pen`** constructors allow programmers to specify the colors and widths of the lines that they wish to draw. The **`System.Drawing`** namespace also provides a **`Pens`** collection containing predefined **`Pen`**s.

Constants in structure `Color` (all are `public static`)	RGB value	Constants in structure `Color` (all are `public static`)	RGB value
`Orange`	255, 200, 0	**`White`**	255, 255, 255
`Pink`	255, 175, 175	**`Gray`**	128, 128, 128
`Cyan`	0, 255, 255	**`DarkGray`**	64, 64, 64
`Magenta`	255, 0, 255	**`Red`**	255, 0, 0
`Yellow`	255, 255, 0	**`Green`**	0, 255, 0
`Black`	0, 0, 0	**`Blue`**	0, 0, 255

Fig. 16.3 `Color` structure **`static`** constants and their RGB values.

Structure `Color` methods and properties	Description
Common Methods	
`static` **`FromArgb`**	Creates a color based on red, green and blue values expressed as **`int`**s from 0 to 255. Overloaded version allows specification of alpha, red, green and blue values.
`static` **`FromName`**	Creates a color from a name, passed as a **`string`**.
Common Properties	
`A`	**`byte`** between 0 and 255, representing the alpha component.
`R`	**`byte`** between 0 and 255, representing the red component.

Fig. 16.4 `Color` structure members (Part 1 of 2.).

Structure `Color` methods and properties	Description
`G`	`byte` between 0 and 255, representing the green component.
`B`	`byte` between 0 and 255, representing the blue component.

Fig. 16.4 `Color` structure members (Part 2 of 2.).

All classes derived from abstract class `Brush` define objects that color the interiors of graphical shapes (for example, the `SolidBrush` constructor takes a `Color` object—the color to draw). In most `Fill` methods, `Brush`es fill a space with a color, pattern or image. Figure 16.5 summarizes various `Brush`es and their functions.

The application in Fig. 16.6 demonstrates several of the methods and properties described in Fig. 16.4. It displays two overlapping rectangles, allowing the user to experiment with color values and color names.

Class	Description
`HatchBrush`	Uses a rectangular brush to fill a region with a pattern. The pattern is defined by a member of the `HatchStyle` enumeration, a foreground color (with which the pattern is drawn) and a background color.
`LinearGradientBrush`	Fills a region with a gradual blend of one color into another. Linear gradients are defined along a line. They can be specified by the two colors, the angle of the gradient and either the width of a rectangle or two points.
`SolidBrush`	Fills a region with one color. Defined by a `Color` object.
`TextureBrush`	Fills a region by repeating a specified `Image` across the surface.

Fig. 16.5 Classes that derive from class `Brush`.

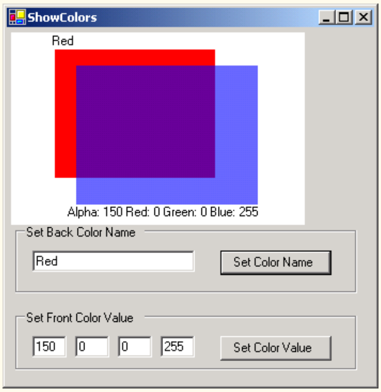
```
// Fig 16.6: ShowColors.cs
// Using different colors in C#.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

```

Fig. 16.6 Color value and alpha demonstration. (Part 1 of 3.)

```
// allows users to change colors using the name of
// the color or argb values
class ShowColors : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;

   // color for back rectangle
   private Color behindColor = Color.Wheat;
   private System.Windows.Forms.GroupBox nameGroup;
   private System.Windows.Forms.GroupBox colorValueGroup;
   private System.Windows.Forms.TextBox colorNameTextBox;
   private System.Windows.Forms.TextBox alphaTextBox;
   private System.Windows.Forms.TextBox redTextBox;
   private System.Windows.Forms.TextBox greenTextBox;
   private System.Windows.Forms.TextBox blueTextBox;
   private System.Windows.Forms.Button colorValueButton;
   private System.Windows.Forms.Button colorNameButton;

   // color for front rectangle
   private Color frontColor =
      Color.FromArgb( 100, 0 , 0, 255 );

   [STAThread]
   static void Main()
   {
      Application.Run( new ShowColors() );
   }

   // Visual Studio .NET generated code

   // override Form OnPaint method
   protected override void OnPaint( PaintEventArgs e )
   {
      Graphics graphicsObject = e.Graphics; // get graphics

      // create text brush
      SolidBrush textBrush = new SolidBrush( Color.Black );

      // create solid brush
      SolidBrush brush = new SolidBrush( Color.White );

      // draw white background
      graphicsObject.FillRectangle( brush, 4, 4, 275, 180 );

      // display name of behindColor
      graphicsObject.DrawString( behindColor.Name, this.Font,
         textBrush, 40, 5 );

      // set brush color and display back rectangle
      brush.Color = behindColor;

      graphicsObject.FillRectangle( brush, 45, 20, 150, 120 );

```

Fig. 16.6 Color value and alpha demonstration. (Part 2 of 3.)

```
      // display Argb values of front color
      graphicsObject.DrawString( "Alpha: " + frontColor.A +
         " Red: " + frontColor.R + " Green: " + frontColor.G
         + " Blue: " + frontColor.B, this.Font, textBrush,
         55, 165 );

      // set brush color and display front rectangle
      brush.Color = frontColor;

      graphicsObject.FillRectangle( brush, 65, 35, 170, 130 );

   } // end method OnPaint

   // handle colorValueButton click event
   private void colorValueButton_Click(
      object sender, System.EventArgs e )
   {
      // obtain new front color from text boxes
      frontColor = Color.FromArgb( Convert.ToInt32(
         alphaTextBox.Text ),
         Convert.ToInt32( redTextBox.Text ),
         Convert.ToInt32( greenTextBox.Text ),
         Convert.ToInt32( blueTextBox.Text ) );

      Invalidate(); // refresh Form
   }

   // handle colorNameButton click event
   private void colorNameButton_Click(
      object sender, System.EventArgs e )
   {
      // set behindColor to color specified in text box
      behindColor = Color.FromName( colorNameTextBox.Text );

      Invalidate(); // refresh Form
   }

} // end class ShowColors
```



Red

Alpha: 150 Red: 0 Green: 0 Blue: 255

Set Back Color Name: Red

Set Front Color Value: 150 0 0 255

Fig. 16.6 Color value and alpha demonstration. (Part 3 of 3.)

When the application begins its execution, it calls class **`ShowColors`**’s **`OnPaint`** method to paint the window. Line 44 gets a reference to **`PaintEventArgs e`**’s **`Graphics`** object and assigns it to **`Graphics`** object **`graphicsObject`**. Lines 47–50 create a black and a white **`SolidBrush`** for drawing on the form. Class **`SolidBrush`** derives from abstract base class **`Brush`**; programmers can draw solid shapes with the **`SolidBrush`**.

`Graphics` method ***`FillRectangle`*** draws a solid white rectangle with the **`Brush`** supplied as a parameter (line 53). It takes as parameters a brush, the *x*- and *y*-coordinates of a point and the width and height of the rectangle to draw. The point represents the upper-left corner of the rectangle. Lines 56–57 display the **`string Name`** property of the **`Brush`**’s **`Color`** property with the **`Graphics DrawString`** method. The programmer has access to several overloaded **`DrawString`** methods; the version demonstrated in lines 56–57 takes a **`string`** to display, the display **`Font`**, a **`Brush`** and the x- and y-coordinates of the location for the **`string`**’s first character.

Lines 60–62 assign the **`Color behindColor`** value to the **`Brush`**’s **`Color`** property and display a rectangle. Lines 65–68 extract and display the ARGB values of **`Color frontColor`** and then display a filled rectangle that overlaps the first.

`Button` event handler **`colorValueButton_Click`** (lines 78–89) uses **`Color`** method **`FromArgb`** to construct a new **`Color`** object from the ARGB values that a user specifies via text boxes. It then assigns the newly created **`Color`** to **`frontColor`**. **`Button`** event handler **`colorNameButton_Click`** (lines 92–99) uses the **`Color`** method **`FromName`** to create a new **`Color`** object from the **`colorName`** that a user enters in a text box. This **`Color`** is assigned to **`behindColor`**.

If the user assigns an alpha value between 0 and 255 for the **`frontColor`**, the effects of alpha blending are apparent. In the screenshot output, the red back rectangle blends with the blue front rectangle to create purple where the two overlap.

Software Engineering Observation 16.1

No methods in class **`Color`** *enable programmers to change the characteristics of the current color. To use a different color, create a new* **`Color`** *object.*

The predefined GUI component ***`ColorDialog`*** is a dialog box that allows users to select from a palette of available colors. It also offers the option of creating custom colors. The program in Fig. 16.7 demonstrates the use of such a dialog. When a user selects a color and presses **OK**, the application retrieves the user’s selection via the **`ColorDialog`**’s ***`Color`*** property.

The GUI for this application contains two **`Button`**s. The top one, **`backgroundColorButton`**, allows the user to change the form and button background colors. The bottom one, **`textColorButton`**, allows the user to change the button text colors.

```
//Fig. 16.7: ShowColorsComplex.cs
// Change the background and text colors of a form.

using System;
using System.Drawing;
using System.Collections;
```

Fig. 16.7 **`ColorDialog`** used to change background and text color. (Part 1 of 3.)

```
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// allows users to change colors using a ColorDialog
public class ShowColorsComplex : System.Windows.Forms.Form
{
   private System.Windows.Forms.Button backgroundColorButton;
   private System.Windows.Forms.Button textColorButton;

   private System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new ShowColorsComplex() );
   }

   // Visual Studio .NET generated code

   // change text color
   private void textColorButton_Click(
      object sender, System.EventArgs e )
   {
      // create ColorDialog object
      ColorDialog colorChooser = new ColorDialog();
      DialogResult result;

      // get chosen color
      result = colorChooser.ShowDialog();

      if ( result == DialogResult.Cancel )
         return;

      // assign forecolor to result of dialog
      backgroundColorButton.ForeColor = colorChooser.Color;
      textColorButton.ForeColor = colorChooser.Color;

   } // end method textColorButton_Click

   // change background color
   private void backgroundColorButton_Click(
      object sender, System.EventArgs e )
   {
      // create ColorDialog object
      ColorDialog colorChooser = new ColorDialog();
      DialogResult result;

      // show ColorDialog and get result
      colorChooser.FullOpen = true;
      result = colorChooser.ShowDialog();

```

Fig. 16.7 `ColorDialog` used to change background and text color. (Part 2 of 3.)

```
59          if ( result == DialogResult.Cancel )
60             return;
61
62          // set background color
63          this.BackColor = colorChooser.Color;
64
65       } // end method backgroundColorButton_Click
66
67    } // end class ShowColorsComplex
```

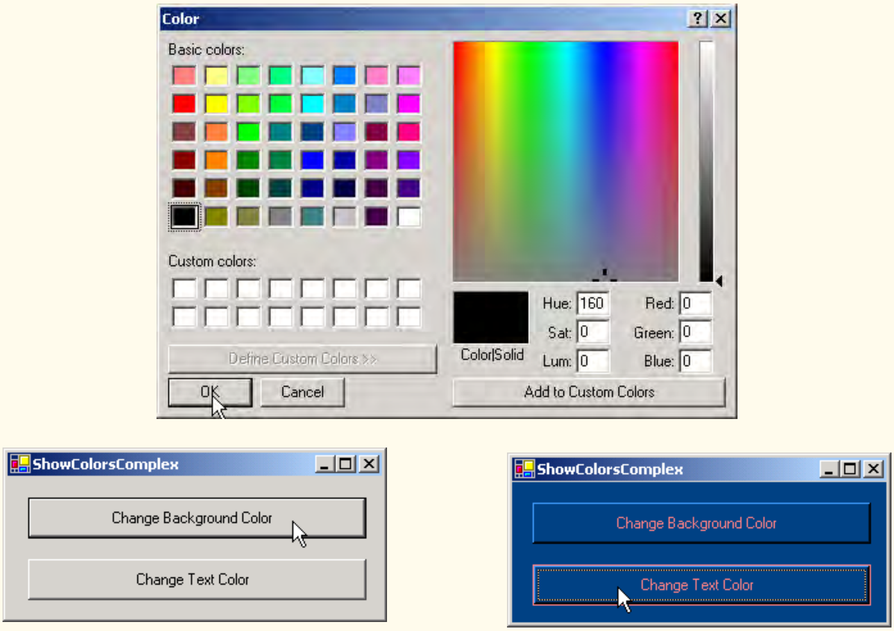

Color
Basic colors:
Custom colors:
Define Custom Colors >>
OK
Cancel
Color|Solid
Hue: 160
Sat: 0
Lum: 0
Red: 0
Green: 0
Blue: 0
Add to Custom Colors
ShowColorsComplex
Change Background Color
Change Text Color
ShowColorsComplex
Change Background Color
Change Text Color

Fig. 16.7 `ColorDialog` used to change background and text color. (Part 3 of 3.)

Lines 28–45 define the event handler that is called when the user clicks **`Button textColorButton`**. The event handler creates a new **`ColorDialog`** named **`colorChooser`** and invokes its **`ShowDialog`** method, which displays the window. Property **`Color`** of **`colorChooser`** stores users' selections. Lines 42–43 set the text color of both buttons to the selected color.

Lines 48–65 define the event handler for button **`backgroundColorButton`**. The method modifies the background color of the form by setting **`BackColor`** equal to the dialog's **`Color`** property. The method creates a new **`ColorDialog`** and sets the dialog's ***`FullOpen`*** property to **`true`**. The dialog now displays all available colors, as shown in the screen capture in Fig. 16.7. The regular color display does not show the right-hand portion of the screen.

Users are not restricted to the **`ColorDialog`**'s 48 colors. To create a custom color, users can click anywhere in the **`ColorDialog`**'s large rectangle—this displays the various color shades. Adjust the slider, hue and other features to refine the color. When finished, click the **Add to Custom Colors** button, which adds the custom color to a square in the custom colors section of the dialog. Clicking **OK** sets the **`Color`** property of the

`ColorDialog` to that color. Selecting a color and pressing the dialog's **OK** button causes the application's background color to change.

16.4 Font Control

This section introduces methods and constants that are related to font control. Once a `Font` has been created, its properties cannot be modified. If programmers require a different `Font`, they must create a new `Font` object—there are many overloaded versions of the `Font` constructor for creating custom `Font`s. Some properties of class `Font` are summarized in Fig. 16.8.

Note that the `Size` property returns the font size as measured in design units, whereas `SizeInPoints` returns the font size as measured in points (the more common measurement). When we say that the `Size` property measures the size of the font in *design units*, we mean that the font size can be specified in a variety of ways, such as inches or millimeters. Some versions of the `Font` constructor accept a `GraphicsUnit` argument—an enumeration that allows users to specify the unit of measurement employed to describe the font size. Members of the `GraphicsUnit` enumeration include `Point` (1/72 inch), `Display` (1/75 inch), `Document` (1/300 inch), `Millimeter`, `Inch` and `Pixel`. If this argument is provided, the `Size` property contains the size of the font as measured in the specified design unit, and the `SizeInPoints` property converts the size of the font into points. For example, if we create a `Font` having size `1` and specify that `GraphicsUnit.Inch` be used to measure the font, the `Size` property will be `1`, and the `SizeInPoints` property will be `72`. If we employ a constructor that does not accept a member of the `GraphicsUnit`, the default measurement for the font size is `GraphicsUnit.Point` (thus, the `Size` and `SizeInPoints` properties will be equal).

Property	Description
`Bold`	Tests a font for a bold font style. Returns `true` if the font is bold.
`FontFamily`	Represents the `FontFamily` of the `Font` (a grouping structure to organize fonts and define their similar properties).
`Height`	Represents the height of the font.
`Italic`	Tests a font for an italic font style. Returns `true` if the font is italic.
`Name`	Represents the font's name as a `string`.
`Size`	Returns a `float` value indicating the current font size measured in design units (design units are any specified units of measurement for the font).
`SizeInPoints`	Returns a `float` value indicating the current font size measured in points.
`Strikeout`	Tests a font for a strikeout font style. Returns `true` if the font is in strikeout format.
`Underline`	Tests a font for a underline font style. Returns `true` if the font is underlined.

Fig. 16.8 `Font` class read-only properties.

Class **`Font`** has a number of constructors. Most require a *font name*, which is a **`string`** representing a font currently supported by the system. Common fonts include Microsoft *SansSerif* and *Serif*. Constructors also usually require the *font size* as an argument. Lastly, **`Font`** constructors usually require a *font style*, specified by the ***`FontStyle`*** enumeration: ***`Bold`***, ***`Italic`***, ***`Regular`***, ***`Strikeout`***, ***`Underline`***. Font styles can be combined via the '|' operator (for example, **`FontStyle.Italic | FontStyle.Bold`**, makes a font both italic and bold).

`Graphics` method ***`DrawString`*** sets the current drawing font—the font in which the text displays—to its **`Font`** argument.

Common Programming Error 16.1

Specifying a font that is not available on a system is a logic error. If this occurs, C# will substitute that system's default font.

The program in Fig. 16.9 displays text in four different fonts, each of a different size. The program uses the **`Font`** constructor to initialize **`Font`** objects (lines 32–47). Each call to the **`Font`** constructor passes a font name (e.g., Arial, Times New Roman, Courier New or Tahoma) as a **`string`**, a font size (a **`float`**) and a **`FontStyle`** object (**`style`**). **`Graphics`** method **`DrawString`** sets the font and draws the text at the specified location. Note that line 29 creates a **`DarkBlue SolidBrush`** object (**`brush`**), causing all **`string`**s drawn with that brush to appear in **`DarkBlue`**.

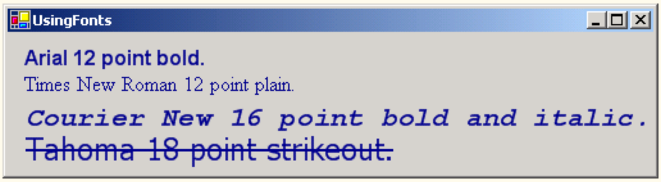
Software Engineering Observation 16.2

*There is no way to change the properties of a **`Font`** object—to use a different font, programmers must create a new **`Font`** object.*

```
 1  // Fig 16.9: UsingFonts.cs
 2  // Demonstrating various font settings.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  // demonstrate font constructors and properties
12  public class UsingFonts : System.Windows.Forms.Form
13  {
14     private System.ComponentModel.Container components = null;
15
16     [STAThread]
17     static void Main()
18     {
19        Application.Run( new UsingFonts() );
20     }
21
22     // Visual Studio .NET generated code
23
```

Fig. 16.9 **`Font`**s and **`FontStyle`**s. (Part 1 of 2.)

```
   // demonstrate various font and style settings
   protected override void OnPaint(
      PaintEventArgs paintEvent )
   {
      Graphics graphicsObject = paintEvent.Graphics;
      SolidBrush brush = new SolidBrush( Color.DarkBlue );

      // arial, 12 pt bold
      FontStyle style = FontStyle.Bold;
      Font arial =
         new Font( new FontFamily( "Arial" ), 12, style );

      // times new roman, 12 pt regular
      style = FontStyle.Regular;
      Font timesNewRoman =
         new Font( "Times New Roman", 12, style );

      // courier new, 16 pt bold and italic
      style = FontStyle.Bold | FontStyle.Italic;
      Font courierNew = new Font( "Courier New", 16, style );

      // tahoma, 18 pt strikeout
      style = FontStyle.Strikeout;
      Font tahoma = new Font( "Tahoma", 18, style );

      graphicsObject.DrawString( arial.Name +
         " 12 point bold.", arial, brush, 10, 10 );

      graphicsObject.DrawString( timesNewRoman.Name +
         " 12 point plain.", timesNewRoman, brush, 10, 30 );

      graphicsObject.DrawString( courierNew.Name +
         " 16 point bold and italic.", courierNew,
         brush, 10, 54 );

      graphicsObject.DrawString( tahoma.Name +
         " 18 point strikeout.", tahoma, brush, 10, 75 );

   } // end method OnPaint

} // end class UsingFonts
```



Arial 12 point bold.

Times New Roman 12 point plain.

Courier New 16 point bold and italic.

~~Tahoma 18 point strikeout.~~

Fig. 16.9 `Font`s and `FontStyle`s. (Part 2 of 2.)

Programmers can define precise information about a font's *metrics* (or properties), such as *height*, *descent* (the amount that characters dip below the baseline), *ascent* (the amount that characters rise above the baseline) and *leading* (the difference between the ascent of one line and the decent of the previous line). Figure 16.10 illustrates these properties.

Class **`FontFamily`** defines characteristics common to a group of related fonts. Class **`FontFamily`** provides several methods used to determine the font metrics that are shared by members of a particular family. These methods are summarized in Fig. 16.11.

The program shown in Fig. 16.12 calls method **`ToString`** to display the metrics of two fonts. Line 32 creates **`Font arial`** and sets it to 12-point Arial font. Line 33 uses class **`Font`** property **`FontFamily`** to obtain object **`arial`**'s **`FontFamily`** object. Lines 38–39 call **`ToString`** to output the **`string`** representation of the font. Lines 41–55 then use methods of class **`FontFamily`** to return integers specifying the ascent, descent, height and leading of the font. Lines 58–77 repeat this process for font **`sansSerif`**, a **`Font`** object derived from the MS Sans Serif **`FontFamily`**.

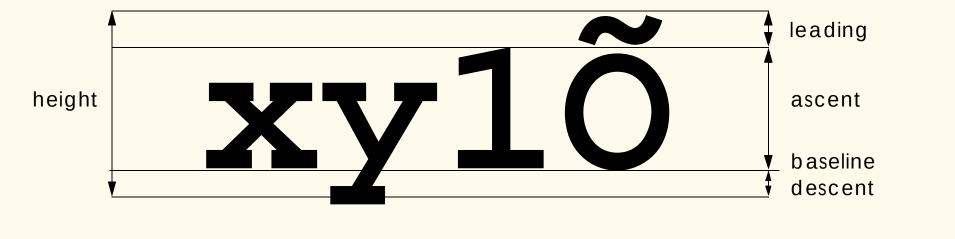


Fig. 16.10 An illustration of font metrics.

Method	Description
`GetCellAscent`	Returns an **`int`** representing the ascent of a font as measured in design units.
`GetCellDescent`	Returns an **`int`** representing the descent of a font as measured in design units.
`GetEmHeight`	Returns an **`int`** representing the height of a font as measured in design points.
`GetLineSpacing`	Returns an **`int`** representing the distance between two consecutive lines of text as measured in design units.

Fig. 16.11 **`FontFamily`** methods that return font-metric information.

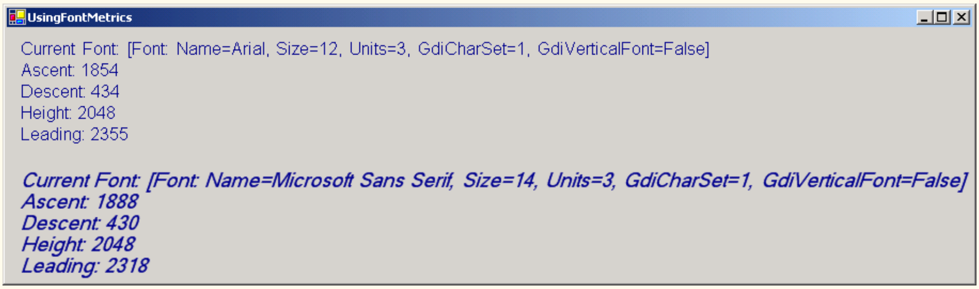
```
1  // Fig 16.12: UsingFontMetrics.cs
2  // Displaying font metric information.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
```

Fig. 16.12 **`FontFamily`** class used to obtain font-metric information. (Part 1 of 3.)

```
using System.Windows.Forms;
using System.Data;

// displays font information
public class UsingFontMetrics : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new UsingFontMetrics() );
   }

   // Visual Studio .NET generated code

   // displays font information
   protected override void OnPaint(
      PaintEventArgs paintEvent )
   {
      Graphics graphicsObject = paintEvent.Graphics;
      SolidBrush brush = new SolidBrush( Color.DarkBlue );

      // Arial font metrics
      Font arial = new Font( "Arial", 12 );
      FontFamily family = arial.FontFamily;
      Font sanSerif = new Font( "Microsoft Sans Serif",
         14, FontStyle.Italic );

      // display Arial font metrics
      graphicsObject.DrawString( "Current Font: " +
         arial.ToString(), arial, brush, 10, 10 );

      graphicsObject.DrawString( "Ascent: " +
         family.GetCellAscent( FontStyle.Regular ), arial,
         brush, 10, 30 );

      graphicsObject.DrawString( "Descent: " +
         family.GetCellDescent( FontStyle.Regular ), arial,
         brush, 10, 50 );

      graphicsObject.DrawString( "Height: " +
         family.GetEmHeight( FontStyle.Regular ), arial,
         brush, 10, 70 );

      graphicsObject.DrawString( "Leading: " +
         family.GetLineSpacing( FontStyle.Regular ), arial,
         brush, 10, 90 );

      // display Sans Serif font metrics
      family = sanSerif.FontFamily;

```

Fig. 16.12 `FontFamily` class used to obtain font-metric information. (Part 2 of 3.)

```
      graphicsObject.DrawString( "Current Font: " +
         sanSerif.ToString(), sanSerif, brush, 10, 130 );

      graphicsObject.DrawString( "Ascent: " +
         family.GetCellAscent( FontStyle.Regular ), sanSerif,
         brush, 10, 150 );

      graphicsObject.DrawString( "Descent: " +
         family.GetCellDescent( FontStyle.Regular ), sanSerif,
         brush, 10, 170 );

      graphicsObject.DrawString( "Height: " +
         family.GetEmHeight( FontStyle.Regular ), sanSerif,
         brush, 10, 190 );

      graphicsObject.DrawString( "Leading: " +
         family.GetLineSpacing( FontStyle.Regular ), sanSerif,
         brush, 10, 210 );

   } // end method OnPaint

} // end class UsingFontMetrics
```



```
Current Font: [Font: Name=Arial, Size=12, Units=3, GdiCharSet=1, GdiVerticalFont=False]
Ascent: 1854
Descent: 434
Height: 2048
Leading: 2355

Current Font: [Font: Name=Microsoft Sans Serif, Size=14, Units=3, GdiCharSet=1, GdiVerticalFont=False]
Ascent: 1888
Descent: 430
Height: 2048
Leading: 2318
```

Fig. 16.12 **`FontFamily`** class used to obtain font-metric information. (Part 3 of 3.)

16.5 Drawing Lines, Rectangles and Ovals

This section presents a variety of **`Graphics`** methods for drawing lines, rectangles and ovals. Each of the drawing methods has several overloaded versions. When employing methods that draw shape outlines, we use versions that take a **`Pen`** and four **`int`**s; when employing methods that draw solid shapes, we use versions that take a **`Brush`** and four **`int`**s. In both instances, the first two **`int`** arguments represent the coordinates of the upper-left corner of the shape or its enclosing area, and the last two **`int`**s indicate the shape's width and height. Figure 16.13 summarizes the **`Graphics`** methods and their parameters.

The application in Fig. 16.14 draws lines, rectangles and ellipses. In this application, we also demonstrate methods that draw filled and unfilled shapes.

Methods ***`DrawRectangle`*** and ***`FillRectangle`*** (lines 33 and 42) draw rectangles on the screen. For each method, the first argument specifies the drawing object to use. The **`DrawRectangle`** method uses a **`Pen`** object, whereas the **`FillRectangle`** method uses a **`Brush`** object (in this case, an instance of **`SolidBrush`**—a class that derives from **`Brush`**). The next two arguments specify the coordinates of the upper-left corner of the *bounding rectangle*, which represents the area in which the rectangle will be drawn. The fourth and fifth arguments specify the rectangle's width and height. Method ***`DrawLine`*** (lines 36–39) takes a **`Pen`** and two pairs of **`int`**s, specifying the start and endpoint of the line. The method then draws a line, using the **`Pen`** object passed to it.

Methods ***`DrawEllipse`*** and ***`FillEllipse`*** each provide overloaded versions that take five arguments. In both methods, the first argument specifies the drawing object to use. The next two arguments specify the upper-left coordinates of the bounding rectangle representing the area in which the ellipse will be drawn. The last two arguments specify the bounding rectangle's width and height, respectively. Figure 16.15 depicts an ellipse bounded by a rectangle. The ellipse touches the midpoint of each of the four sides of the bounding rectangle. The bounding rectangle is not displayed on the screen.

Graphics Drawing Methods and Descriptions.

Note: Many of these methods are overloaded—consult the documentation for a full listing.

`DrawLine( Pen p, int x1, int y1, int x2, int y2 )`
Draws a line from (**`x1`**, **`y1`**) to (**`x2`**, **`y2`**). The **`Pen`** determines the color, style and width of the line.

`DrawRectangle( Pen p, int x, int y, int width, int height )`
Draws a rectangle of the specified width and height. The top-left corner of the rectangle is at point (**`x`**, **`y`**). The **`Pen`** determines the color, style, and border width of the rectangle.

`FillRectangle( Brush b, int x, int y, int width, int height )`
Draws a solid rectangle of the specified width and height. The top-left corner of the rectangle is at point (**`x`**, **`y`**). The **`Brush`** determines the fill pattern inside the rectangle.

`DrawEllipse( Pen p, int x, int y, int width, int height )`
Draws an ellipse inside a rectangle. The width and height of the rectangle are as specified, and its top-left corner is at point (**`x`**, **`y`**). The **`Pen`** determines the color, style and border width of the ellipse.

`FillEllipse( Brush b, int x, int y, int width, int height )`
Draws a filled ellipse inside a rectangle. The width and height of the rectangle are as specified, and its top-left corner is at point (**`x`**, **`y`**). The **`Brush`** determines the pattern inside the ellipse.

Fig. 16.13 **`Graphics`** methods that draw lines, rectangles and ovals.

```
1  // Fig. 16.14: LinesRectanglesOvals.cs
2  // Demonstrating lines, rectangles and ovals.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
```

Fig. 16.14 Demonstration of methods that draw lines, rectangles and ellipses. (Part 1 of 3.)

```
using System.Windows.Forms;
using System.Data;

// draws shapes on the Form
public class LinesRectanglesOvals : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new LinesRectanglesOvals() );
   }

   // Visual Studio .NET generated code

   protected override void OnPaint(
      PaintEventArgs paintEvent )
   {
      // get graphics object
      Graphics g = paintEvent.Graphics;
      SolidBrush brush = new SolidBrush( Color.Blue );
      Pen pen = new Pen( Color.AliceBlue );

      // create filled rectangle
      g.FillRectangle( brush, 90, 30, 150, 90 );

      // draw lines to connect rectangles
      g.DrawLine( pen, 90, 30, 110, 40 );
      g.DrawLine( pen, 90, 120, 110, 130 );
      g.DrawLine( pen, 240, 30, 260, 40 );
      g.DrawLine( pen, 240, 120, 260, 130 );

      // draw top rectangle
      g.DrawRectangle( pen, 110, 40, 150, 90 );

      // set brush to red
      brush.Color = Color.Red;

      // draw base Ellipse
      g.FillEllipse( brush, 280, 75, 100, 50 );

      // draw connecting lines
      g.DrawLine( pen, 380, 55, 380, 100 );
      g.DrawLine( pen, 280, 55, 280, 100 );

      // draw Ellipse outline
      g.DrawEllipse( pen, 280, 30, 100, 50 );

   } // end method OnPaint

} // end class LinesRectanglesOvals
```

Fig. 16.14 Demonstration of methods that draw lines, rectangles and ellipses. (Part 2 of 3.)

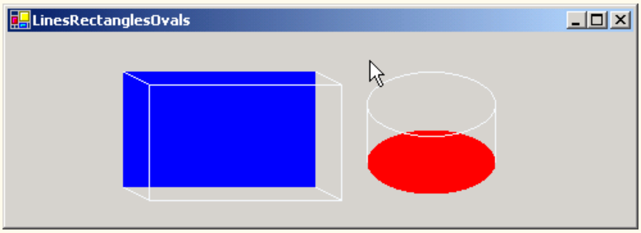


Fig. 16.14 Demonstration of methods that draw lines, rectangles and ellipses. (Part 3 of 3.)

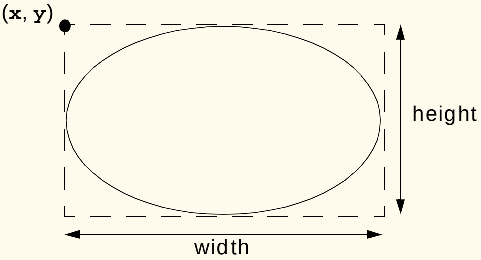


Fig. 16.15 Ellipse bounded by a rectangle.

16.6 Drawing Arcs

*Arc*s are portions of ellipses and are measured in degrees, beginning at a *starting angle* and continuing for a specified number of degrees called the *arc angle*. An arc is said to *sweep* (traverse) its arc angle, beginning from its starting angle. Arcs that sweep in a clockwise direction are measured in positive degrees, whereas arcs that sweep in a counterclockwise direction are measured in negative degrees. Figure 16.16 depicts two arcs. Note that the left portion of the figure sweeps downward from zero degrees to approximately 110 degrees. Similarly, the arc in the right portion of the figure sweeps upward from zero degrees to approximately –110 degrees.

Notice the dashed boxes around the arcs in Fig. 16.16. We draw each arc as part of an oval (the rest of which is not visible). When drawing an oval, we specify the oval's dimensions in the form of a bounding rectangle that encloses the oval. The boxes in Fig. 16.16 correspond to these bounding rectangles. The **`Graphics`** methods used to draw arcs—**`DrawArc`**, **`DrawPie`** and **`FillPie`**—are summarized in Fig. 16.17.

The program in Fig. 16.18 draws six images (three arcs and three filled pie slices) to demonstrate the arc methods listed in Fig. 16.17. To illustrate the bounding rectangles that determine the sizes and locations of the arcs, the arcs are displayed inside red rectangles that have the same *x*-coordinates, *y*-coordinates and width and height arguments as those that define the bounding rectangles for the arcs.

Lines 28–35 create the objects that we need to draw various arcs: **`Graphics`** objects, **`Rectangle`**s, **`SolidBrush`**es and **`Pen`**s. Lines 38–39 then draw a rectangle and an arc inside the rectangle. The arc sweeps 360 degrees, forming a circle. Line 42 changes the location of the **`Rectangle`** by setting its **`Location`** property to a new **`Point`**. The **`Point`** constructor takes the *x*- and *y*-coordinates of the new point. The **`Location`** property determines the upper-left corner of the **`Rectangle`**. After drawing the rectangle, the program draws an arc that starts at 0 degrees and sweeps 110 degrees. Because angles in C# increase in a clockwise direction, the arc sweeps downward.

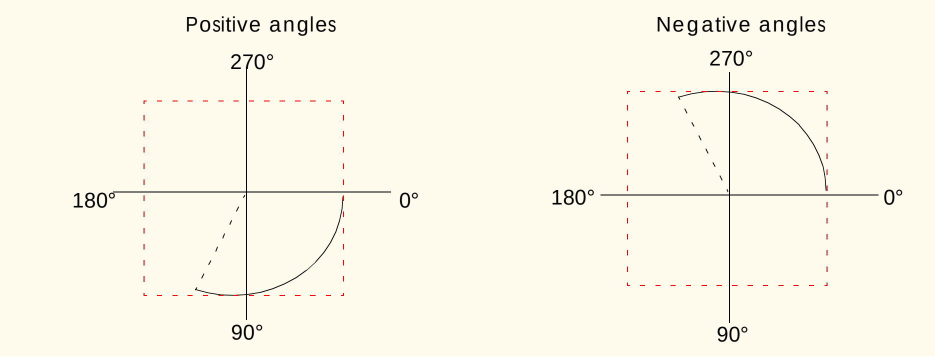


Fig. 16.16 Positive and negative arc angles.

Graphics Methods And Descriptions

Note: Many of these methods are overloaded—consult the documentation for a complete listing.

```
DrawArc( Pen p, int x, int y, int width, int height,
   int startAngle, int sweepAngle )
```

Draws an arc of an ellipse, beginning from angle **`startAngle`** (in degrees) and sweeping **`sweepAngle`** degrees. The ellipse is defined by a bounding rectangle of width **`w`**, height **`h`** and upper-left corner (**`x`**,**`y`**). The **`Pen`** determines the color, border width and style of the arc.

```
DrawPie( Pen p, int x, int y, int width, int height,
   int startAngle, int sweepAngle )
```

Draws a pie section of an ellipse, beginning from angle **`startAngle`** (in degrees) and sweeping **`sweepAngle`** degrees. The ellipse is defined by a bounding rectangle of width **`w`**, height **`h`** and upper-left corner (**`x`**,**`y`**). The **`Pen`** determines the color, border width and style of the arc.

```
FillPie( Brush b, int x, int y, int width, int height,
   int startAngle, int sweepAngle )
```

Functions similarly to **`DrawPie`**, except draws a solid arc (i.e., a sector). The **`Brush`** determines the fill pattern for the solid arc.

Fig. 16.17 **`Graphics`** methods for drawing arcs.

```
// Fig. 16.18: DrawArcs.cs
// Drawing various arcs on a form.
```

Fig. 16.18 Arc-method demonstration. (Part 1 of 3.)

```

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// draws various arcs
public class DrawArcs : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new DrawArcs() );
   }

   // Visual Studio .NET generated code

   private void DrawArcs_Paint(
      object sender, System.Windows.Forms.PaintEventArgs e )
   {
      // get graphics object
      Graphics graphicsObject = e.Graphics;
      Rectangle rectangle1 =
         new Rectangle( 15, 35, 80, 80 );
      SolidBrush brush1 =
         new SolidBrush( Color.Firebrick );
      Pen pen1 = new Pen( brush1, 1 );
      SolidBrush brush2 = new SolidBrush( Color.DarkBlue );
      Pen pen2 = new Pen( brush2, 1 );

      // start at 0 and sweep 360 degrees
      graphicsObject.DrawRectangle( pen1, rectangle1 );
      graphicsObject.DrawArc( pen2, rectangle1, 0, 360 );

      // start at 0 and sweep 110 degrees
      rectangle1.Location = new Point( 100, 35 );
      graphicsObject.DrawRectangle( pen1, rectangle1 );
      graphicsObject.DrawArc( pen2, rectangle1, 0, 110 );

      // start at 0 and sweep -270 degrees
      rectangle1.Location = new Point( 185, 35 );
      graphicsObject.DrawRectangle( pen1, rectangle1 );
      graphicsObject.DrawArc( pen2, rectangle1, 0, -270 );

      // start at 0 and sweep 360 degrees
      rectangle1.Location = new Point( 15, 120 );
      rectangle1.Size = new Size( 80, 40 );
      graphicsObject.DrawRectangle( pen1, rectangle1 );
      graphicsObject.FillPie( brush2, rectangle1, 0, 360 );
```

Fig. 16.18 Arc-method demonstration. (Part 2 of 3.)

```
56
57        // start at 270 and sweep -90 degrees
58        rectangle1.Location = new Point( 100, 120 );
59        graphicsObject.DrawRectangle( pen1, rectangle1 );
60        graphicsObject.FillPie(
61           brush2, rectangle1, 270, -90 );
62
63        // start at 0 and sweep -270 degrees
64        rectangle1.Location = new Point( 185, 120 );
65        graphicsObject.DrawRectangle( pen1, rectangle1 );
66        graphicsObject.FillPie(
67           brush2, rectangle1, 0, -270 );
68
69     } // end method DrawArcs_Paint
70
71  } // end class DrawArcs
```

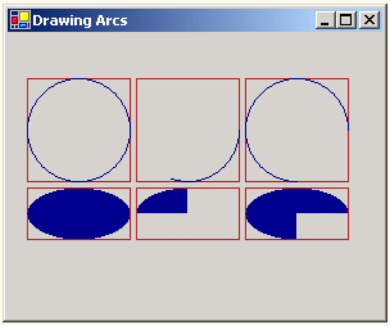


Fig. 16.18 Arc-method demonstration. (Part 3 of 3.)

Lines 47–49 perform similar functions, except that the specified arc sweeps -270 degrees. The **Size** property of a **Rectangle** determines the arc's height and width. Line 53 sets the **Size** property to a new **Size** object, which changes the size of the rectangle.

The remainder of the program is similar to the portions described above, except that a **SolidBrush** is used with method **FillPie**. The resulting arcs, which are filled, can be seen in the bottom half of the screenshot Fig. 16.18.

16.7 Drawing Polygons and Polyli]nes

Polygons are multisided shapes. There are several **Graphics** methods used to draw polygons: **DrawLines** draws a series of connected points, **DrawPolygon** draws a closed polygon and **FillPolygon** draws a solid polygon. These methods are described in Fig. 16.19. The program in Fig. 16.20 allows users to draw polygons and connected lines via the methods listed in Fig. 16.19.

To allow the user to specify a variable number of points, line 26 declares **ArrayList points** as a container for our **Point** objects. Lines 29–31 declare the **Pen** and **Brush** used to color our shapes. The **MouseDown** event handler (lines 42–49) for **Panel drawPanel** stores mouse-click locations in the **points ArrayList**. It then calls method **Invalidate** of **drawPanel** to ensure that the panel refreshes to accommodate the new point. Method **drawPanel_Paint** (lines 51–82) handles the **Panel**'s **Paint** event. It obtains the panel's **`Graphics`** object (line 55) and, if the **`ArrayList points`** contains two or more **`Points`**, displays the polygon with the method that the user selected via the GUI radio buttons (lines 58–80). In lines 61–63, we extract an **`Array`** from the **`ArrayList`** via method **`ToArray`**. Method **`ToArray`** can take a single argument to determine the type of the returned array; we obtain the type from the first element in the **`ArrayList`**.

Method	Description
`DrawLines`	Draws a series of connected lines. The coordinates of each point are specified in an array of **`Point`**s. If the last point is different from the first point, the figure is not closed.
`DrawPolygon`	Draws a polygon. The coordinates of each point are specified in an array of **`Point`** objects. This method draws a closed polygon, even if the last point is different from the first point.
`FillPolygon`	Draws a solid polygon. The coordinates of each point are specified in an array of **`Points`**. This method draws a closed polygon, even if the last point is different from the first point.

Fig. 16.19 **`Graphics`** methods for drawing polygons.

```
1  // Fig. 16.20: DrawPolygons.cs
2  // Demonstrating polygons.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Data;
10
11 public class PolygonForm : System.Windows.Forms.Form
12 {
13    private System.Windows.Forms.Button colorButton;
14    private System.Windows.Forms.Button clearButton;
15    private System.Windows.Forms.GroupBox typeGroup;
16    private System.Windows.Forms.RadioButton
17       filledPolygonOption;
18    private System.Windows.Forms.RadioButton lineOption;
19    private System.Windows.Forms.RadioButton polygonOption;
20    private System.Windows.Forms.Panel drawPanel;
21
22    private
23       System.ComponentModel.Container components = null;
24
25    // contains list of polygon vertices
26    private ArrayList points = new ArrayList();
27
```

Fig. 16.20 Polygon-drawing demonstration. (Part 1 of 4.)

```
   // initialize default pen and brush
   Pen pen = new Pen( Color.DarkBlue );

   SolidBrush brush = new SolidBrush( Color.DarkBlue );

   [STAThread]
   static void Main()
   {
      Application.Run( new PolygonForm() );
   }

   // Visual Studio .NET generated code

   // draw panel mouse down event handler
   private void drawPanel_MouseDown(
      object sender, System.Windows.Forms.MouseEventArgs e )
   {
      // add mouse position to vertex list
      points.Add( new Point( e.X, e.Y ) );
      drawPanel.Invalidate(); // refresh panel

   } // end method drawPanel_MouseDown

   private void drawPanel_Paint(
      object sender, System.Windows.Forms.PaintEventArgs e )
   {
      // get graphics object for panel
      Graphics graphicsObject = e.Graphics;

      // if arraylist has 2 or more points, display shape
      if ( points.Count > 1 )
      {
         // get array for use in drawing functions
         Point[] pointArray =
            ( Point[] )points.ToArray(
            points[ 0 ].GetType() );

         if ( polygonOption.Checked )

            // draw polygon
            graphicsObject.DrawPolygon( pen, pointArray );

         else if ( lineOption.Checked )

            // draw lines
            graphicsObject.DrawLines( pen, pointArray );

         else if ( filledPolygonOption.Checked )

            // draw filled
            graphicsObject.FillPolygon(
               brush, pointArray );
      }
```

Fig. 16.20 Polygon-drawing demonstration. (Part 2 of 4.)

```

      } // end method drawPanel_Paint

      // handle clearButton click event
      private void clearButton_Click(
         object sender, System.EventArgs e )
      {
         points = new ArrayList(); // remove points

         drawPanel.Invalidate(); // refresh panel

      } // end method clearButton_Click

      // handle polygon radio button CheckedChanged event
      private void polygonOption_CheckedChanged(
         object sender, System.EventArgs e)
      {
         drawPanel.Invalidate(); // refresh panel

      } // end method polygonOption_CheckedChanged

      // handle line radio button CheckedChanged event
      private void lineOption_CheckedChanged(
         object sender, System.EventArgs e)
      {
         drawPanel.Invalidate(); // refresh panel

      } // end method lineOption_CheckedChanged

      // handle filled polygon radio button
      // CheckedChanged event
      private void filledPolygonOption_CheckedChanged(
         object sender, System.EventArgs e)
      {
         drawPanel.Invalidate(); // refresh panel

      } // end method filledPolygonOption_CheckedChanged

      // handle colorButton click event
      private void colorButton_Click(
         object sender, System.EventArgs e)
      {
         // create new color dialog
         ColorDialog dialogColor = new ColorDialog();

         // show dialog and obtain result
         DialogResult result = dialogColor.ShowDialog();

         // return if user cancels
         if ( result == DialogResult.Cancel )
            return;

         pen.Color = dialogColor.Color;   // set pen to color
```

Fig. 16.20 Polygon-drawing demonstration. (Part 3 of 4.)

```
134          brush.Color = dialogColor.Color; // set brush
135          drawPanel.Invalidate();          // refresh panel;
136
137       } // end method colorButton_Click
138
139    } // end class PolygonForm
```

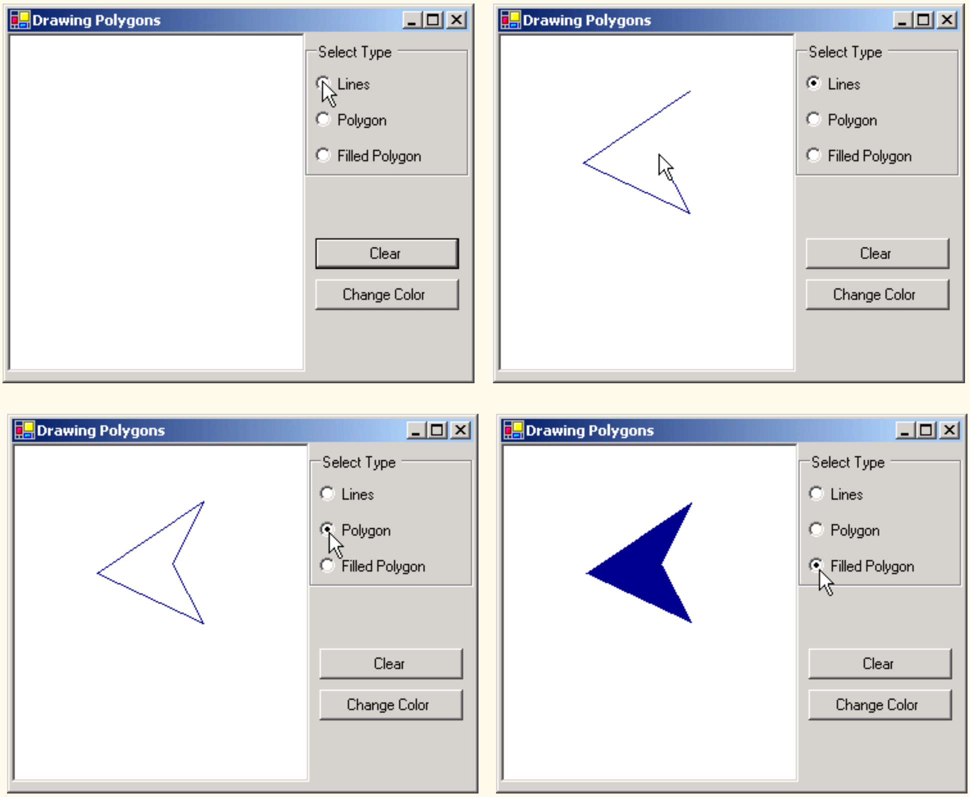


Fig. 16.20 Polygon-drawing demonstration. (Part 4 of 4.)

Method **`clearButton_Click`** (lines 85–92) handles the **Clear** button's click event, creates an empty **`ArrayList`** (causing the old list to be erased) and refreshes the display. Lines 95–117 define the event handlers for the radio buttons' **`CheckedChanged`** event. Each method refreshes **`Panel drawPanel`** to ensure that the panel display reflects the selected drawing type. Event method **`colorButton_Click`** (120–137) allows the user to select a new drawing color with a **`ColorDialog`**, using the same technique demonstrated in Fig. 16.7.

16.8 Advanced Graphics Capabilities

C# offers many additional graphics capabilities. The **`Brush`** hierarchy, for example, also includes ***`HatchBrush`***, ***`LinearGradientBrush`***, ***`PathGradientBrush`*** and ***`TextureBrush`***.

The program in Fig. 16.21 demonstrates several graphics features, such as dashed lines, thick lines and the ability to fill shapes with patterns. These represent just a few of the additional capabilities of the **`System.Drawing`** namespace.

```
// Fig. 16.21: DrawShapes.cs
// Drawing various shapes on a form.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Drawing.Drawing2D;

// draws shapes with different brushes
public class DrawShapesForm : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new DrawShapesForm() );
   }

   // Visual Studio .NET generated code

   // draw various shapes on form
   private void DrawShapesForm_Paint(
      object sender, System.Windows.Forms.PaintEventArgs e )
   {
      // references to object we will use
      Graphics graphicsObject = e.Graphics;

      // ellipse rectangle and gradient brush
      Rectangle drawArea1 =
         new Rectangle( 5, 35, 30, 100 );
      LinearGradientBrush linearBrush =
         new LinearGradientBrush( drawArea1, Color.Blue,
         Color.Yellow, LinearGradientMode.ForwardDiagonal );

      // pen and location for red outline rectangle
      Pen thickRedPen = new Pen( Color.Red, 10 );
      Rectangle drawArea2 = new Rectangle( 80, 30, 65, 100 );

      // bitmap texture
      Bitmap textureBitmap = new Bitmap( 10, 10 );

      // get bitmap graphics
      Graphics graphicsObject2 =
         Graphics.FromImage( textureBitmap );
```

Fig. 16.21 Shapes drawn on a form. (Part 1 of 3.)

```

         // brush and pen used throughout program
         SolidBrush solidColorBrush =
            new SolidBrush( Color.Red );
         Pen coloredPen = new Pen( solidColorBrush );

         // draw ellipse filled with a blue-yellow gradient
         graphicsObject.FillEllipse(
            linearBrush, 5, 30, 65, 100 );

         // draw thick rectangle outline in red
         graphicsObject.DrawRectangle( thickRedPen, drawArea2 );

         // fill textureBitmap with yellow
         solidColorBrush.Color = Color.Yellow;
         graphicsObject2.FillRectangle(
            solidColorBrush, 0, 0, 10, 10 );

         // draw small black rectangle in textureBitmap
         coloredPen.Color = Color.Black;
         graphicsObject2.DrawRectangle(
            coloredPen, 1, 1, 6, 6 );

         // draw small blue rectangle in textureBitmpa
         solidColorBrush.Color = Color.Blue;
         graphicsObject2.FillRectangle(
            solidColorBrush, 1, 1, 3, 3 );

         // draw small red square in textureBitmap
         solidColorBrush.Color = Color.Red;
         graphicsObject2.FillRectangle(
            solidColorBrush, 4, 4, 3, 3 );

         // create textured brush and
         // display textured rectangle
         TextureBrush texturedBrush =
            new TextureBrush( textureBitmap );
         graphicsObject.FillRectangle(
            texturedBrush, 155, 30, 75, 100 );

         // draw pie-shaped arc in white
         coloredPen.Color = Color.White;
         coloredPen.Width = 6;
         graphicsObject.DrawPie(
            coloredPen, 240, 30, 75, 100, 0, 270 );

         // draw lines in green and yellow
         coloredPen.Color = Color.Green;
         coloredPen.Width = 5;
         graphicsObject.DrawLine(
            coloredPen, 395, 30, 320, 150 );

```

Fig. 16.21 Shapes drawn on a form. (Part 2 of 3.)

```
101         // draw a rounded, dashed yellow line
102         coloredPen.Color = Color.Yellow;
103         coloredPen.DashCap = ( DashCap )LineCap.Round;
104         coloredPen.DashStyle = DashStyle.Dash;
105         graphicsObject.DrawLine(
106            coloredPen, 320, 30, 395, 150 );
107
108      } // end method DrawShapesForm_Paint
109
110   } // end class DrawShapesForm
```

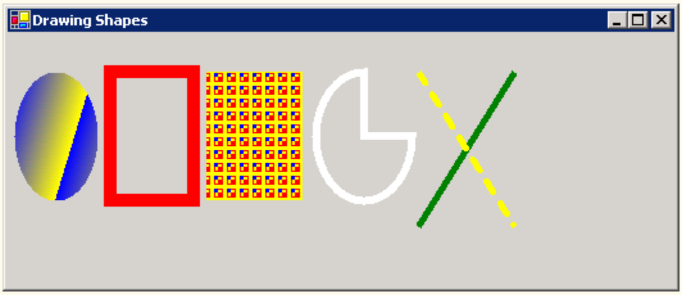


Fig. 16.21 Shapes drawn on a form. (Part 3 of 3.)

Lines 26–108 define the **`Paint`** event handler for our form. Lines 35–37 create **`LinearGradientBrush`** object **`linearBrush`**, which resides in the **`System.Drawing.Drawing2D`** namespace. A **`LinearGradientBrush`** enables users to draw with a color gradient. The **`LinearGradientBrush`** used in this example takes four arguments: A **`Rectangle`**, two **`Color`**s and a member of enumeration **`LinearGradientMode`**. In C#, all linear gradients are defined along a line that determines the gradient endpoint. This line can be specified either by starting and ending points or by the diagonal of a rectangle. The first argument, **`Rectangle drawArea1`**, specifies the defining line for **`LinearGradientBrush linearBrush`**. This **`Rectangle`** argument represents the endpoints of the linear gradient—the upper-left corner is the starting point, and the bottom-right corner is the ending point. The second and third arguments specify the colors that the gradient will use. In this case, the color of the ellipse will gradually change from **`Color.Blue`** to **`Color.Yellow`**. The last argument, a type from the enumeration ***`LinearGradientMode`***, specifies the linear gradient's direction. In our case, we use ***`LinearGradientMode.ForwardDiagonal`***, which creates a gradient from the upper-left to the lower-right corner. We then use **`Graphics`** method **`FillEllipse`** in lines 56–57 to draw an ellipse with **`linearBrush`**; the color gradually changes from blue to yellow, as described above.

In line 40, we create a **`Pen`** object **`thickRedPen`**. We pass to **`thickRedPen`**'s constructor **`Color.Red`** and **`int`** argument **`10`**, indicating that we want **`thickRedPen`** to draw red lines that are 10 pixels wide.

Line 44 creates a new ***`Bitmap`*** image, which initially is empty. Class **`Bitmap`** can produce images in color and gray scale; this particular **`Bitmap`** is 10 pixels wide and 10 pixels tall. Method ***`FromImage`*** (line 47–48) is a **`static`** member of class **`Graphics`** and retrieves the **`Graphics`** object associated with an **`Image`**, which may be used to draw on an image. Lines 63–80 draw on the **`Bitmap`** a pattern consisting of black, blue, red and yellow rectangles and lines. A `TextureBrush` is a brush that fills the interior of a shape with an image, rather than a solid color. In line 86–87, `TextureBrush` object `textureBrush` fills a rectangle with our `Bitmap`. The `TextureBrush` constructor version that we use takes as an argument an image that defines its texture.

Next, we draw a pie-shaped arc with a thick white line. Lines 90–91 set `coloredPen`'s color to `White` and modify its width to be six pixels. We then draw the pie on the form by specifying the `Pen` the *x*-coordinate, *y*-coordinate, length and width of the bounding rectangle and the start angle and sweep angle.

Finally, lines 103–104 make use of `System.Drawing.Drawing2D` enumerations ***`DashCap`*** and ***`DashStyle`*** to draw a diagonal dashed line. Line 103 sets the ***`DashCap`*** property of `coloredPen` (not to be confused with the `DashCap` enumeration) to a member of the `DashCap` enumeration. The `DashCap` enumeration specifies the styles for the start and end of a dashed line. In this case, we want both ends of the dashed line to be rounded, so we use ***`DashCap.Round`***. Line 104 sets the ***`DashStyle`*** property of `coloredPen` (not to be confused with the `DashStyle` enumeration) to ***`DashStyle.Dash`***, indicating that we want our line to consist entirely of dashes.

Our next example demonstrates the use of a *general path*. A general path is a shape constructed from straight lines and complex curves. An object of class ***`GraphicsPath`*** (`System.Drawing.Drawing2D` namespace) represents a general path. The `GraphicsPath` class provides functionality that enables the creation of complex shapes from vector-based primitive graphics objects. A `GraphicsPath` object consists of figures defined by simple shapes. The start point of each vector-graphics object (such as a line or arc) that is added to the path is connected by a straight line to the end point of the previous object. When called, the `CloseFigure` method attaches the final vector-graphic object endpoint to the initial starting point for the current figure by a straight line, then starts a new figure. Method `StartFigure` begins a new figure within the path without closing the previous figure.

The program of Fig. 16.22 draws general paths in the shape of five-pointed stars. Line 45 sets the origin of the `Graphics` object. The arguments to method `TranslateTransform` indicate that the origin should be translated to the coordinates (150, 150). Lines 36–39 define two `int` arrays, representing the *x*- and *y*-coordinates of the points in the star, and line 42 defines `GraphicsPath` object `star`. A `for` loop then creates lines to connect the points of the star and adds these lines to `star`. We use `GraphicsPath` method ***`AddLine`*** to append a line to the shape. The arguments of `AddLine` specify the coordinates for the line's endpoints; each new call to `AddLine` adds a line from the previous point to the current point. Line 38 uses `GraphicsPath` method ***`CloseFigure`*** to complete the shape.

```
1   // Fig. 16.22: DrawStarsForm.cs
2   // Using paths to draw stars on the form.
3
4   using System;
5   using System.Drawing;
6   using System.Collections;
7   using System.ComponentModel;
```

Fig. 16.22 Paths used to draw stars on a form. (Part 1 of 3.)

```
using System.Windows.Forms;
using System.Data;
using System.Drawing.Drawing2D;

// draws randomly colored stars
public class DrawStarsForm : System.Windows.Forms.Form
{
   private
      System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new DrawStarsForm() );
   }

   // Visual Studio .NET generated code

   // create path and draw stars along it
   private void DrawStarsForm_Paint(
      object sender, System.Windows.Forms.PaintEventArgs e )
   {
      Graphics graphicsObject = e.Graphics;
      Random random = new Random();
      SolidBrush brush =
         new SolidBrush( Color.DarkMagenta );

      // x and y points of the path
      int[] xPoints =
         { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
      int[] yPoints =
         { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };

      // create graphics path for star;
      GraphicsPath star = new GraphicsPath();

      // translate the origin to (150, 150)
      graphicsObject.TranslateTransform( 150, 150 );

      // create star from series of points
      for ( int i = 0; i <= 8; i += 2 )
         star.AddLine( xPoints[ i ], yPoints[ i ],
            xPoints[ i + 1 ], yPoints[ i + 1 ] );

      // close the shape
      star.CloseFigure();

      // rotate the origin and draw stars in random colors
      for ( int i = 1; i <= 18; i++ )
      {
         graphicsObject.RotateTransform( 20 );

```

Fig. 16.22 Paths used to draw stars on a form. (Part 2 of 3.)

```
60             brush.Color = Color.FromArgb(
61                random.Next( 200, 255 ), random.Next( 255 ),
62                random.Next( 255 ), random.Next( 255 ) );
63
64             graphicsObject.FillPath( brush, star );
65          }
66
67       } // end method DrawStarsForm_Paint
68
69    } // end class DrawStarsForm
```

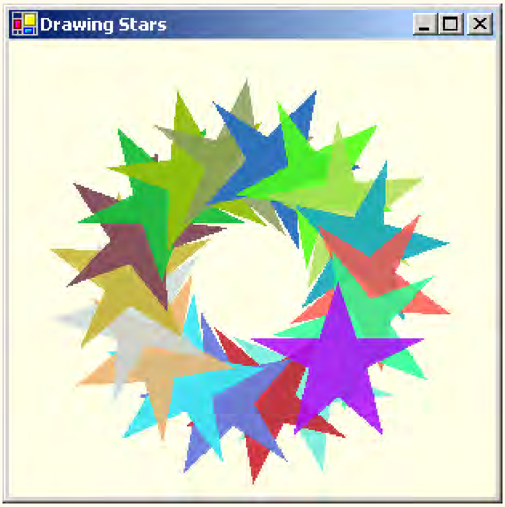


Fig. 16.22 Paths used to draw stars on a form. (Part 3 of 3.)

The **`for`** structure in lines 56–65 draws the **`star`** 18 times, rotating it around the origin. Line 58 uses **`Graphics`** method ***`RotateTransform`*** to move to the next position on the form; the argument specifies the rotation angle in degrees. **`Graphics`** method **`FillPath`** (line 64) then draws a filled version of the **`star`** with the **`Brush`** created on lines 60–62. The application determines the **`SolidBrush`**'s color randomly, using **`Random`** method **`Next`**.

16.9 Introduction to Multimedia

C# offers many convenient ways to include images and animations in programs. People who entered the computing field decades ago used computers primarily to perform arithmetic calculations. As the discipline evolves, we are beginning to realize the importance of computers' data-manipulation capabilities. We are seeing a wide variety of exciting new three-dimensional applications. Multimedia programming is an entertaining and innovative field, but one that presents many challenges

Multimedia applications demand extraordinary computing power. Until recently, affordable computers with this amount of power were not available. However, today's ultrafast processors are making multimedia-based applications commonplace. As the market for multimedia explodes, users are purchasing the faster processors, larger memories and wider communications bandwidths needed to support multimedia applications. This benefits the computer and communications industries, which provide the hardware, software and services fueling the multimedia revolution.

In the remaining sections of this chapter, we introduce the use and manipulation of images and other multimedia features and capabilities. Section 16.10 discusses how to load, display and scale images; Section 16.11 demonstrates image animation; Section 16.12 presents the video capabilities of the Windows Media Player control; and Section 16.13 explores Microsoft Agent technology.

16.10 Loading, Displaying and Scaling Images

C#'s multimedia capabilities include graphics, images, animations and video. Previous sections demonstrated C#'s vector-graphics capabilities; this section concentrates on image manipulation. The Windows form that we create in Fig. 16.23 demonstrates the loading of an `Image` (`System.Drawing` namespace). The application allows users to enter a desired height and width for the `Image`, which then is displayed in the specified size.

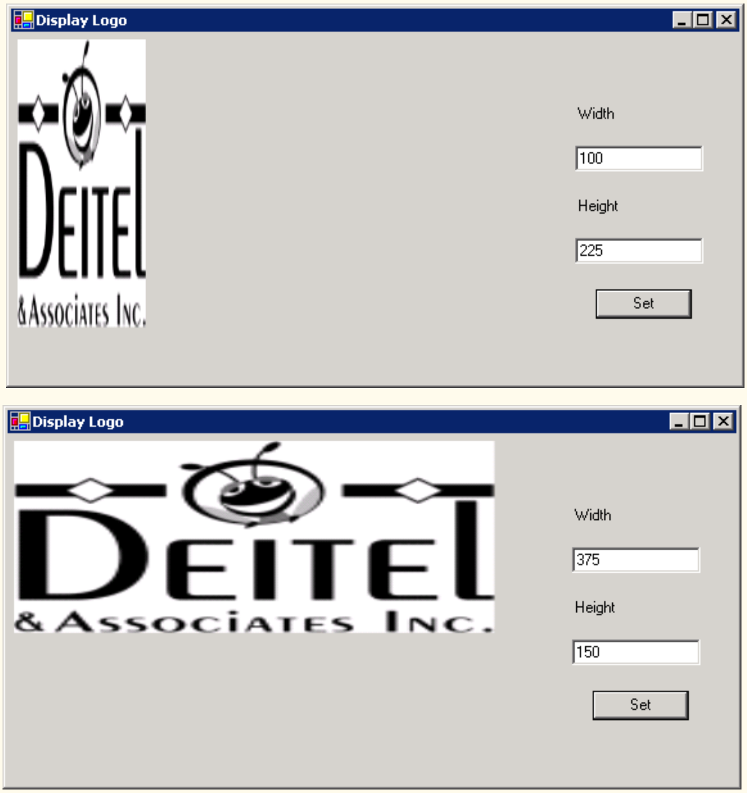
Lines 23–24 declare `Image` reference `image`. The `static` `Image` method *`FromFile`* then retrieves an image stored on disk and assigns it to `image` (line 24). Line 31 uses `Form` method *`CreateGraphics`* to create a `Graphics` object associated with the `Form`; we use this object to draw on the `Form`. Method `CreateGraphics` is inherited from class `Control`; all Windows controls, such as `Button`s and `Panel`s, also provide this method. When users click **Set**, the width and height parameters are validated to ensure that they are not too large. If the parameters are valid, line 59 calls `Graphics` method *`Clear`* to paint the entire `Form` in the current background color. Lines 62–63 call `Graphics` method *`DrawImage`* with the following parameters: the image to draw, the *x*-coordinate of the upper-left corner, the *y*-coordinate of the upper-left corner, the width of the image and the height of the image. If the width and height do not correspond to the image's original dimensions, the image is scaled to fit the new specifications.

```
 1  // Fig. 16.23: DisplayLogoForm.cs
 2  // Displaying and resizing an image.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  // displays an image and allows the user to resize it
12  public class DisplayLogoForm : System.Windows.Forms.Form
13  {
14     private System.Windows.Forms.Button setButton;
15     private System.Windows.Forms.TextBox heightTextBox;
16     private System.Windows.Forms.Label heightLabel;
```

Fig. 16.23 Image resizing. (Part 1 of 3.)

```
      private System.Windows.Forms.TextBox widthTextBox;
      private System.Windows.Forms.Label widthLabel;

      private
         System.ComponentModel.Container components = null;

      private
         Image image = Image.FromFile( "images/Logo.gif" );
      private Graphics graphicsObject;

      public DisplayLogoForm()
      {
         InitializeComponent();

         graphicsObject = this.CreateGraphics();
      }

      [STAThread]
      static void Main()
      {
         Application.Run( new DisplayLogoForm() );
      }

      // Visual Studio .NET generated code

      private void setButton_Click(
         object sender, System.EventArgs e )
      {
         // get user input
         int width = Convert.ToInt32( widthTextBox.Text );
         int height = Convert.ToInt32( heightTextBox.Text );

         // if dimensions specified are too large
         // display problem
         if ( width > 375 || height > 225 )
         {
            MessageBox.Show( "Height or Width too large" );

            return;
         }

         // clear Windows Form
         graphicsObject.Clear( this.BackColor );

         // draw image
         graphicsObject.DrawImage(
            image, 5, 5, width, height );

      } // end method setButton_Click

   } // end class DisplayLogoForm
```

Fig. 16.23 Image resizing. (Part 2 of 3.)



Width
100
Height
225

Width
375
Height
150

Fig. 16.23 Image resizing. (Part 3 of 3.)

16.11 Animating a Series of Images

The next example animates a series of images stored in an array. The application uses the same techniques to load and display **`Image`**s as those illustrated in Fig. 16.23. The images were created with Adobe Photoshop.

The animation in Fig. 16.24 uses a **`PictureBox`**, which contains the images that we animate. We use a **`Timer`** to cycle through the images, causing a new image to display every 50 milliseconds. Variable **`count`** keeps track of the current image number and increases by one every time we display a new image. The array includes 30 images (numbered 0–29); when the application reaches image 29, it returns to image 0. The 30 images were prepared in advance and placed in the **`images`** folder inside the **`bin/Debug`** directory of the project.

```
// Fig. 16.24: LogoAnimator.cs
// Program that animates a series of images.

```

Fig. 16.24 Animation of a series of images. (Part 1 of 3.)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// animates a series of 30 images
public class LogoAnimator : System.Windows.Forms.Form
{
   private System.Windows.Forms.PictureBox logoPictureBox;
   private System.Windows.Forms.Timer Timer;
   private System.ComponentModel.IContainer components;

   private ArrayList images = new ArrayList();
   private int count = -1;

   public LogoAnimator()
   {
      InitializeComponent();

      for ( int i = 0; i < 30; i++ )
         images.Add( Image.FromFile( "images/deitel" + i +
            ".gif" ) );

      // load first image
      logoPictureBox.Image = ( Image ) images[ 0 ];

      // set PictureBox to be the same size as Image
      logoPictureBox.Size = logoPictureBox.Image.Size;

   } // end constructor

   [STAThread]
   static void Main()
   {
      Application.Run( new LogoAnimator() );
   }

   // Visual Studio .NET generated code

   private void Timer_Tick(
      object sender, System.EventArgs e )
   {
      // increment counter
      count = ( count + 1 ) % 30;

      // load next image
      logoPictureBox.Image = ( Image )images[ count ];

   } // end method Timer_Tick

} // end class LogoAnimator
```

Fig. 16.24 Animation of a series of images. (Part 2 of 3.)

LogoAnimator
LogoAnimator
LogoAnimator

Fig. 16.24 Animation of a series of images. (Part 3 of 3.)

Lines 25–27 load each of 30 images and place them in an ***`ArrayList`***. **`ArrayList`** method ***`Add`*** allows us to add objects to the **`ArrayList`**; we use this method in lines 26–27 to add each **`Image`**. Line 30 places the first image in the **`PictureBox`**, using the **`ArrayList`** indexer. Line 33 modifies the size of the **`PictureBox`** so that it is equal to the size of the **`Image`** it is displaying. The event handler for **`timer`**'s **`Tick`** event (line 45–54) then displays the next image from the **`ArrayList`**.

Performance Tip 16.2

It is more efficient to load an animation's frames as one image than to load each image separately. (A painting program, such as Adobe Photoshop®, Jasc® or Paint Shop Pro™, can be used to combine the animation's frames into one image.) If the images are being loaded separately from the Web, each loaded image requires a separate connection to the site on which the images are stored; this process can result in poor performance.

Performance Tip 16.3

Loading animation frames can cause program delays, because the program waits for all frames to load before displaying them.

The following chess example demonstrates the capabilities of GDI+ as they pertain to a chess-game application. These include techniques for two-dimensional *collision detection*, the selection of single frames from a multi-frame image and *regional invalidation* (refreshing only the required parts of the screen) to increase performance. Two-dimensional collision detection is the detection of an overlap between two shapes. In the next example, we demonstrate the simplest form of collision detection, which determines whether a point (the mouse-click location) is contained within a rectangle (a chess-piece image).

Class **`ChessPiece`** (Fig. 16.25) is a container class for the individual chess pieces. Lines 11–19 define a public enumeration of constants that identify each chess-piece type. The constants also serve to identify the location of each piece in the chess-piece image file. **`Rectangle`** object **`targetRectangle`** (lines 25–26) identifies the image location on the chess board. The **`x`** and **`y`** properties of the rectangle are assigned in the **`ChessPiece`** constructor, and all chess-piece images have height and width **`75`**.

The **`ChessPiece`** constructor (lines 29–40) requires that the calling class define a chess-piece type, its **`x`** and **`y`** location and the **`Bitmap`** containing all chess-piece images. Rather than loading the chess-piece image within the class, we allow the calling class to pass the image. This avoids the image-loading overhead for each piece. It also increases the flexibility of the class by allowing the user to change images; for example, in this case, we use the class for both black and white chess-piece images. Lines 37–39 extract a subimage that contains only the current piece's bitmap data. Our chess-piece images are defined in a specific manner: One image contains six chess-piece images, each defined within a 75-pixel block, resulting in a total image size of 450-by-75. We obtain a single image via **`Bitmap`**'s **`Clone`** method, which allows us to specify a rectangle image location and the desired pixel format. The location is a 75-by-75 pixel block with its upper-left corner **`x`** equal to **`75 * type`** and the corresponding **`y`** equal to **`0`**. For the pixel format, we specify constant **`DontCare`**, causing the format to remain unchanged.

Method **`Draw`** (lines 43–46) causes the **`ChessPiece`** to draw **`pieceImage`** in **`targetRectangle`** on the passed **`Graphics`** object. **`Method GetBounds`** returns the object **`targetRectangle`** for use in collision detection, and **`SetLocation`** allows the calling class to specify a new piece location.

```
 1  // Fig. 16.25 : ChessPiece.cs
 2  // Storage class for chess piece attributes.
 3
 4  using System;
 5  using System.Drawing;
 6
 7  // represents a chess piece
 8  public class ChessPiece
 9  {
10     // define chess-piece type constants
11     public enum Types
12     {
13        KING,
14        QUEEN,
15        BISHOP,
16        KNIGHT,
17        ROOK,
18        PAWN
19     }
20
21     private int currentType; // this object's type
22     private Bitmap pieceImage; // this object's image
23
24     // default display location
25     private Rectangle targetRectangle =
26        new Rectangle( 0, 0, 75, 75 );
27
28     // construct piece
29     public ChessPiece( int type, int xLocation,
30        int yLocation, Bitmap sourceImage )
31     {
32        currentType = type; // set current type
33        targetRectangle.X = xLocation; // set current x location
34        targetRectangle.Y = yLocation; // set current y location
35
36        // obtain pieceImage from section of sourceImage
37        pieceImage = sourceImage.Clone(
38           new Rectangle( type * 75, 0, 75, 75 ),
39           System.Drawing.Imaging.PixelFormat.DontCare );
40     }
```

Fig. 16.25 Container class for chess pieces. (Part 1 of 2.)

```

   // draw chess piece
   public void Draw( Graphics graphicsObject )
   {
      graphicsObject.DrawImage( pieceImage, targetRectangle );
   }

   // obtain this piece's location rectangle
   public Rectangle GetBounds()
   {
      return targetRectangle;
   } // end method GetBounds

   // set this piece's location
   public void SetLocation( int xLocation, int yLocation )
   {
      targetRectangle.X = xLocation;
      targetRectangle.Y = yLocation;

   } // end method SetLocation

} // end class ChessPiece
```

Fig. 16.25 Container class for chess pieces. (Part 2 of 2.)

Class **`ChessGame`** (Fig. 16.26) defines the game and graphics code for our chess game. Lines 23–33 define class-scope variables the program requires. **`ArrayList chessTile`** (line 23) stores the board tile images. It contains four images: Two light tiles and two dark tiles (to increase board variety). **`ArrayList chessPieces`** (line 26) stores all active **`ChessPiece`** objects and **`int selectedIndex`** (line 29) identifies the index in **`chessPieces`** of the currently selected piece. The **`board`** (line 30) is an 8-by-8, two-dimensional **`int`** array corresponding to the squares of a chess board. Each board element is an integer from 0 to 3 that corresponds to an index in **`chessTile`** and is used to specify the chess-board-square image. **`const int TILESIZE`** (line 33) defines the size of each tile in pixels.

The chess game GUI consists of **`Form ChessGame`**, the area in which we draw the tiles; **`Panel pieceBox`**, the window in which we draw the pieces (note that **`pieceBox`** background color is set to **`"transparent"`**); and a **`Menu`** that allows the user to begin a new game. Although the pieces and tiles could have been drawn on the same form, doing so would decrease performance. We would be forced to refresh the board as well as the pieces every time we refreshed the control.

The **`ChessGame Load`** event (lines 44–56) loads each tile image into **`chessTile`**. It then calls method **`ResetBoard`** to refresh the **`Form`** and begin the game. Method **`ResetBoard`** (lines 59–169) assigns **`chessPieces`** to a new **`ArrayList`**, loading images for both the black and the white chess-piece sets, and creates **`Bitmap selected`** to define the currently selected **`Bitmap`** set. Lines 82–167 loop through 64 positions on the chess board, setting the tile color and piece for each tile. Lines 86–87 cause the currently selected image to switch to the **`blackPieces`** after the fifth row. If the row counter is on the first or last row, lines 94–134 add a new piece to **`chessPieces`**. The type of the piece is based on the current column we are initializing. Pieces in chess are positioned in the following order, from left to right: Rook, knight, bishop, queen, king, bishop, knight and rook. Lines 137–146 add a new pawn at the current location if the current **`row`** is second or seventh.

A chess board is defined by alternating light and dark tiles across a row in a pattern where the color that starts each row is equal to the color of the last tile of the previous row. Lines 151–162 assign the current board-tile color as an index in the **`board`** array. Based on the alternating value of **`bool`** variable **`light`** and the results of the random operation on line 149, **`0`** and **`1`** are light tiles, whereas **`2`** and **`3`** are dark tiles. Line 166 inverts the value of **`light`** at the end of each row to maintain the staggered effect of a chess board.

```
 1  // Fig. 16.26: ChessGame.cs
 2  // Chess Game graphics code.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  // allows 2 players to play chess
12  public class ChessGame : System.Windows.Forms.Form
13  {
14     private System.Windows.Forms.PictureBox pieceBox;
15     private System.Windows.Forms.MainMenu GameMenu;
16     private System.Windows.Forms.MenuItem gameItem;
17     private System.Windows.Forms.MenuItem newGameItem;
18
19     private
20        System.ComponentModel.Container components = null;
21
22     // ArrayList for board tile images
23     ArrayList chessTile = new ArrayList();
24
25     // ArrayList for chess pieces
26     ArrayList chessPieces = new ArrayList();
27
28     // define index for selected piece
29     int selectedIndex = -1;
30     int[,] board = new int[ 8, 8 ]; // board array
31
32     // define chess tile size in pixels
33     private const int TILESIZE = 75;
34
35     [STAThread]
36     static void Main()
37     {
38        Application.Run( new ChessGame() );
39     }
40
41     // Visual Studio .NET generated code
42
```

Fig. 16.26 Chess-game code. (Part 1 of 8.)

```
   // load tile bitmaps and reset game
   private void ChessGame_Load(
      object sender, System.EventArgs e)
   {
      // load chess board tiles
      chessTile.Add( Bitmap.FromFile( "lightTile1.png" ) );
      chessTile.Add( Bitmap.FromFile( "lightTile2.png" ) );
      chessTile.Add( Bitmap.FromFile( "darkTile1.png" ) );
      chessTile.Add( Bitmap.FromFile( "darkTile2.png" ) );

      ResetBoard(); // initialize board
      Invalidate(); // refresh form

   } // end method ChessGame_Load

   // initialize pieces to start and rebuild board
   private void ResetBoard()
   {
      int current = -1;
      ChessPiece piece;
      Random random = new Random();
      bool light = false;
      int type;

      // ensure empty arraylist
      chessPieces = new ArrayList();

      // load whitepieces image
      Bitmap whitePieces =
         ( Bitmap )Image.FromFile( "whitePieces.png" );

      // load blackpieces image
      Bitmap blackPieces =
         ( Bitmap )Image.FromFile( "blackPieces.png" );

      // set whitepieces drawn first
      Bitmap selected = whitePieces;

      // traverse board rows in outer loop
      for ( int row = 0;
         row <= board.GetUpperBound( 0 ); row++ )
      {
         // if at bottom rows, set to black pieces images
         if ( row > 5 )
            selected = blackPieces;

         // traverse board columns in inner loop
         for ( int column = 0;
            column <= board.GetUpperBound( 1 ); column++ )
         {
            // if first or last row, organize pieces
            if ( row == 0 || row == 7 )
            {
```

Fig. 16.26 Chess-game code. (Part 2 of 8.)

```
                  switch( column )
                  {
                     case 0:
                     case 7: // set current piece to rook
                        current =
                           ( int )ChessPiece.Types.ROOK;
                        break;

                     case 1:
                     case 6: // set current piece to knight
                        current =
                           ( int )ChessPiece.Types.KNIGHT;
                        break;

                     case 2:
                     case 5: // set current piece to bishop
                        current =
                           ( int )ChessPiece.Types.BISHOP;
                        break;

                     case 3: // set current piece to king
                        current =
                           ( int )ChessPiece.Types.KING;
                        break;

                     case 4: // set current piece to queen
                        current =
                           ( int )ChessPiece.Types.QUEEN;
                        break;
                  }

                  // create current piece at start position
                  piece = new ChessPiece( current,
                     column * TILESIZE, row * TILESIZE,
                     selected );

                  // add piece to arraylist
                  chessPieces.Add( piece );
               }

               // if second or seventh row, organize pawns
               if ( row == 1 || row == 6 )
               {
                  piece = new ChessPiece(
                     ( int )ChessPiece.Types.PAWN,
                     column * TILESIZE, row * TILESIZE,
                     selected );

                  // add piece to arraylist
                  chessPieces.Add( piece );
               }

```

Fig. 16.26 Chess-game code. (Part 3 of 8.)

```
            // determine board piece type
            type = random.Next( 0, 2 );

            if ( light )
            {
               // set light tile
               board[ row, column ] = type;
               light = false;
            }
            else
            {
               // set dark tile
               board[ row, column ] = type + 2;
               light = true;
            }
         }

         // account for new row tile color switch
         light = !light;
      }

   } // end method ResetBoard

   // display board in form OnPaint event
   private void ChessGame_Paint(
      object sender, System.Windows.Forms.PaintEventArgs e)
   {
      // obtain graphics object
      Graphics graphicsObject = e.Graphics;

      for ( int row = 0;
         row <= board.GetUpperBound( 0 ); row++ )
      {
         for ( int column = 0;
            column <= board.GetUpperBound( 1 ); column++ )
         {
            // draw image specified in board array
            graphicsObject.DrawImage(
               (Image)chessTile[ board[ row, column ] ],
               new Point( TILESIZE * column,
               TILESIZE * row ) );
         }
      }

   } // end method ChessGame_Paint

   // return index of piece that intersects point
   // optionally exclude a value
   private int CheckBounds( Point point, int exclude )
   {
      Rectangle rectangle; // current bounding rectangle

```

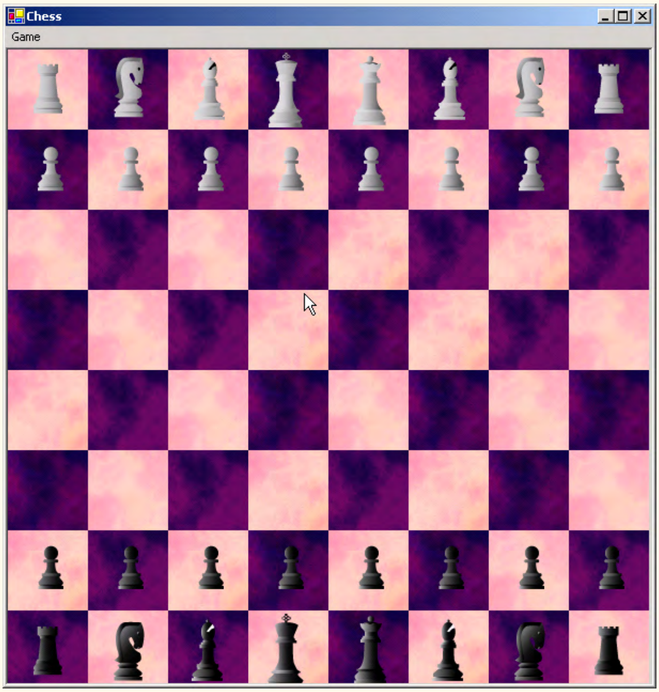
Fig. 16.26 Chess-game code. (Part 4 of 8.)

```
         for ( int i = 0; i < chessPieces.Count; i++ )
         {
            // get piece rectangle
            rectangle = GetPiece( i ).GetBounds();

            // check if rectangle contains point
            if ( rectangle.Contains( point ) && i != exclude )
               return i;
         }

         return -1;

      } // end method CheckBounds

      // handle pieceBox paint event
      private void pieceBox_Paint(
         object sender, System.Windows.Forms.PaintEventArgs e)
      {
         // draw all pieces
         for ( int i = 0; i < chessPieces.Count; i++ )
            GetPiece( i ).Draw( e.Graphics );

      } // end method pieceBox_Paint

      private void pieceBox_MouseDown(
         object sender, System.Windows.Forms.MouseEventArgs e)
      {
         // determine selected piece
         selectedIndex =
            CheckBounds( new Point( e.X, e.Y ), -1 );

      } // end method pieceBox_MouseDown

      // if piece is selected, move it
      private void pieceBox_MouseMove(
         object sender, System.Windows.Forms.MouseEventArgs e)
      {
         if ( selectedIndex > -1 )
         {
            Rectangle region = new Rectangle(
               e.X - TILESIZE * 2, e.Y - TILESIZE * 2,
               TILESIZE * 4, TILESIZE * 4 );

            // set piece center to mouse
            GetPiece( selectedIndex ).SetLocation(
               e.X - TILESIZE / 2, e.Y - TILESIZE / 2 );

            // refresh immediate are
            pieceBox.Invalidate( region );
         }

      } // end method pieceBox_MouseMove
```

Fig. 16.26 Chess-game code. (Part 5 of 8.)

```

   // on mouse up deselect piece and remove taken piece
   private void pieceBox_MouseUp(
      object sender, System.Windows.Forms.MouseEventArgs e)
   {
      int remove = -1;

      //if chess piece was selected
      if ( selectedIndex > -1 )
      {
         Point current = new Point( e.X, e.Y );
         Point newPoint = new Point(
            current.X - ( current.X % TILESIZE ),
            current.Y - ( current.Y % TILESIZE ) );

         // check bounds with point, exclude selected piece
         remove = CheckBounds( current, selectedIndex );

         // snap piece into center of closest square
         GetPiece( selectedIndex ).SetLocation( newPoint.X,
            newPoint.Y );

         // deselect piece
         selectedIndex = -1;

         // remove taken piece
         if ( remove > -1 )
            chessPieces.RemoveAt( remove );
      }

      // refresh pieceBox to ensure artifact removal
      pieceBox.Invalidate();

   } // end method pieceBox_MouseUp

   // helper function to convert
   // ArrayList object to ChessPiece
   private ChessPiece GetPiece( int i )
   {
      return (ChessPiece)chessPieces[ i ];
   } // end method GetPiece

   // handle NewGame menu option click
   private void newGameItem_Click(
      object sender, System.EventArgs e)
   {
      ResetBoard(); // reinitialize board
      Invalidate(); // refresh form

   } // end method newGameItem_Click

} // end class ChessGame
```

Fig. 16.26 Chess-game code. (Part 6 of 8.)



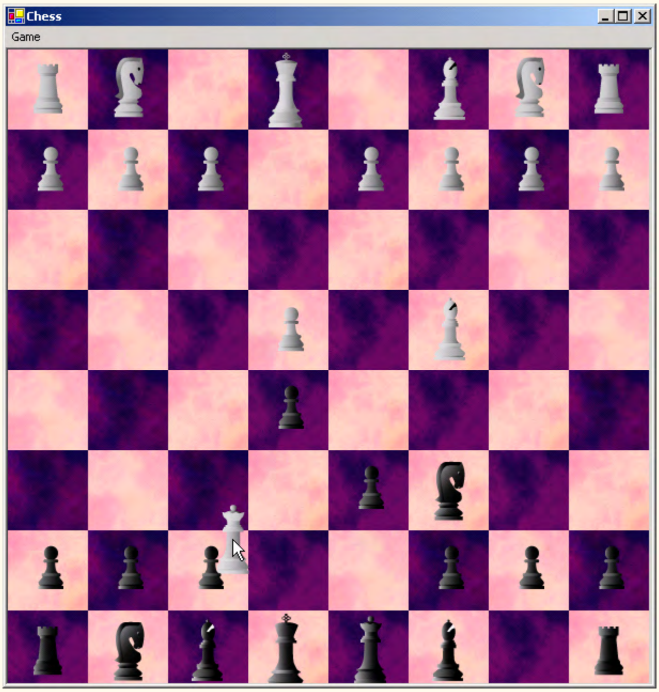


Fig. 16.26 Chess-game code. (Part 7 of 8.)

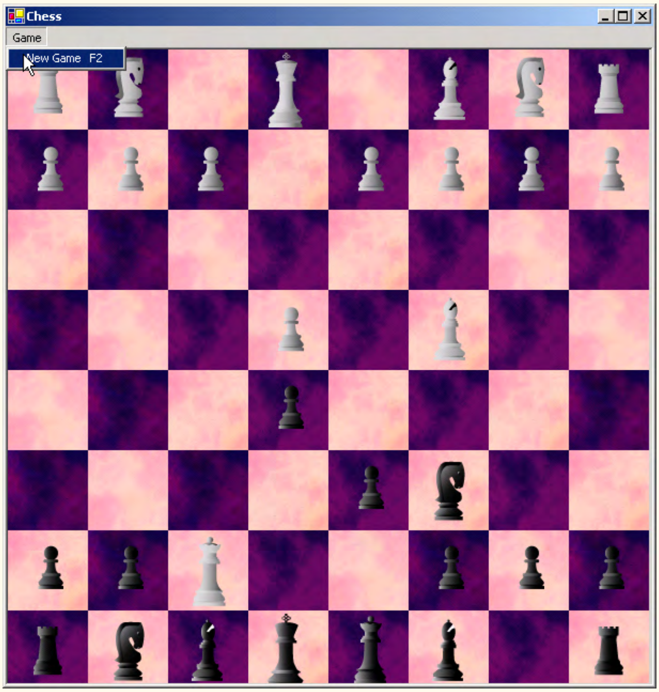


Fig. 16.26 Chess-game code. (Part 8 of 8.)

Method **`ChessGame_Paint`** (lines 172–192) handles this class **`Form`**'s **`Paint`** event and draws the tiles according to their values in the board array. Method **`pieceBox_Paint`**, which handles the **`pieceBox Panel paint`** event, iterates through each element of the **`chessPiece ArrayList`** and calls its **`Draw`** method.

The **`MouseDown`** event handler (lines 224–231) calls method **`CheckBounds`** with the location of the user's click to determine whether the user selected a piece. **`CheckBounds`** returns an integer locating a collision from a given point.

The **`MouseMove`** event handler (lines 234–251) moves the currently selected piece with the mouse. Lines 244–245 set the selected piece location to the mouse-cursor position, adjusting the location by up to half a tile to center the image on the mouse. Lines 239–241 define and refresh a region of the **`Panel`** that spans two tiles in every direction from the mouse. As mentioned earlier in the chapter, the **`Invalidate`** method is slow. This means that the **`MouseMove`** event handler might be called again several times before the **`Invalidate`** method completes. If a user working on a slow computer moves the mouse quickly, the application could leave behind *artifacts*. An artifact is any unintended visual abnormality in a graphical program. By causing the program to refresh a two-square rectangle, which should suffice in most cases, we achieve a significant performance enhancement over an entire component refresh during each **`MouseMove`** event.

Lines 254–285 define the **`MouseUp`** event handler. If a piece has been selected, lines 260–280 determine the index in **`chessPieces`** of any piece collision, remove the collided piece, snap (align) the current piece into a valid location and deselect the piece. We check for piece collisions to allow the chess piece to "take" other chess pieces. Line 268 checks whether any piece (excluding the currently selected piece) is beneath the current mouse location. If a collision is detected, the returned piece index is assigned to `int remove`. Lines 271–272 determine the closest valid chess tile and “snap” the selected piece to that location. If `remove` contains a positive value, line 279 removes the object at that index from the `chessPieces` `ArrayList`. Finally, the entire `Panel` is `Invalidate`d in line 283 to display the new piece location and remove any artifacts created during the move.

Method `CheckBounds` (lines 196–212) is a collision-detection helper method; it iterates through the `chessPieces` `ArrayList` and returns the index of any piece rectangle containing the point value passed to the method (the mouse location, in this example). Method `CheckBounds` optionally can exclude a single piece index (to ignore the selected index in the `MouseUp` event handler, in this example).

Lines 289–292 define helper function `GetPiece`, which simplifies the conversion from `object`s in the `ArrayList` `chessPieces` to `ChessPiece` types. Method `newGameItem_Click` handles the `NewGame` menu item click event, calls `RefreshBoard` to reset the game and `Invalidate`s the entire form.

16.12 Windows Media Player

The Windows Media Player control enables an application to play video and sound in many multimedia formats. These include MPEG (Motion Pictures Experts Group) audio and video, AVI (audio-video interleave) video, WAV (Windows wave-file format) audio and MIDI (Musical Instrument Digital Interface) audio. Users can find preexisting audio and video on the Internet, or they can create their own files, using available sound and graphics packages.

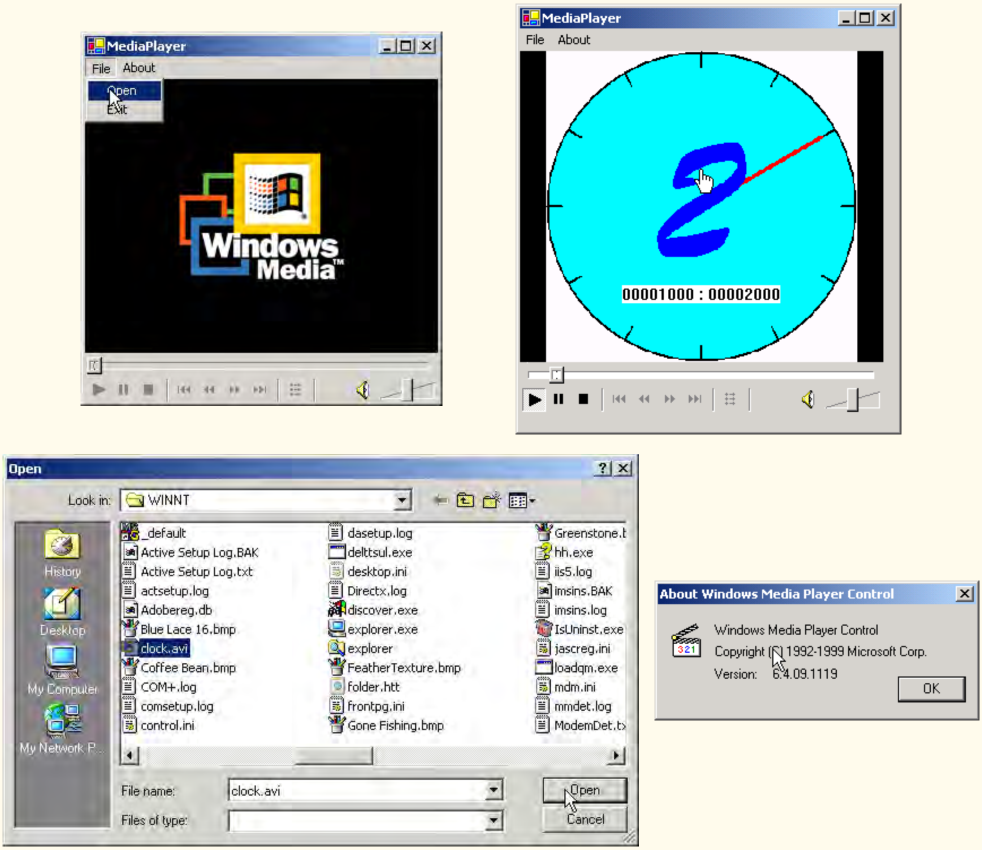
The application in Fig. 16.27 demonstrates the Windows Media Player control, which enables users to play multimedia files. To use the Windows Media Player control, programmers must add the control to the **Toolbox**. This is accomplished by first selecting **Customize Toolbox** from the **Tool** menu to display the **Customize Toolbox** dialog box. In the dialog box, scroll down and select the option **Windows Media Player**. Then, click the **OK** button to dismiss the dialog box. The icon for the Windows Media Player control now should appear at the bottom of the **Toolbox**.

```
// Fig 16.27: MediaPlayerTest.cs
// Demonstrates the Windows Media Player control

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// allows users to play media files using a
// Windows Media Player control
public class MediaPlayer : System.Windows.Forms.Form
{
   private System.Windows.Forms.MainMenu applicationMenu;
```

Fig. 16.27 Windows Media Player demonstration. (Part 1 of 3.)

```
   private System.Windows.Forms.MenuItem fileItem;
   private System.Windows.Forms.MenuItem openItem;
   private System.Windows.Forms.MenuItem exitItem;
   private System.Windows.Forms.MenuItem aboutItem;
   private System.Windows.Forms.MenuItem aboutMessageItem;
   private System.Windows.Forms.OpenFileDialog
      openMediaFileDialog;
   private AxMediaPlayer.AxMediaPlayer player;

   private
      System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new MediaPlayer() );
   }

   // Visual Studio .NET generated code

   // open new media file in Windows Media Player
   private void openItem_Click(
      object sender, System.EventArgs e)
   {
      openMediaFileDialog.ShowDialog();

      player.FileName = openMediaFileDialog.FileName;

      // adjust the size of the Media Player control and
      // the Form according to the size of the image
      player.Size = new Size( player.ImageSourceWidth,
         player.ImageSourceHeight );

      this.Size = new Size( player.Size.Width + 20,
         player.Size.Height + 60 );

   } // end method openItem_Click

   private void exitItem_Click(
      object sender, System.EventArgs e)
   {
      Application.Exit();

   } // end method exitItem_Click

   private void aboutMessageItem_Click(
      object sender, System.EventArgs e)
   {
      player.AboutBox();

   } // end method aboutMessageItem_Click

} // end class MediaPlayer
```

Fig. 16.27 Windows Media Player demonstration. (Part 2 of 3.)


MediaPlayer
File
About
Open
Exit
Windows Media
MediaPlayer
File
About
00001000 : 00002000
Open
Look in: WINNT
History
Desktop
My Computer
My Network P...
_default
Active Setup Log.BAK
Active Setup Log.txt
actsetup.log
Adobereg.db
Blue Lace 16.bmp
clock.avi
Coffee Bean.bmp
COM+.log
comsetup.log
control.ini
dasetup.log
delttsul.exe
desktop.ini
Directx.log
discover.exe
explorer.exe
explorer
FeatherTexture.bmp
folder.htt
frontpg.ini
Gone Fishing.bmp
Greenstone.b
hh.exe
iis5.log
imsins.BAK
imsins.log
IsUninst.exe
jascreg.ini
loadqm.exe
mdm.ini
mmdet.log
ModemDet.tx
Open
Cancel
About Windows Media Player Control
OK

File name: clock.avi

Files of type:

Windows Media Player Control
Copyright (C) 1992-1999 Microsoft Corp.
Version: 6.4.09.1119

Fig. 16.27 Windows Media Player demonstration. (Part 3 of 3.)

The Windows Media Player control provides several buttons that allow the user to play the current file, pause, stop, play the previous file, rewind, forward and play the next file. The control also includes a volume control and trackbars to select a specific position in the media file.

The application provides a **`MainMenu`**, which includes **File** and **About** menus. The **File** menu contains the **Open** and **Exit** menu items; the **About** menu contains the **About Windows Media Player** menu item.

When a user chooses **Open** from the **File** menu, event handler **`openItem_Click`** (lines 37–52) executes. An **`OpenFileDialog`** box displays (line 40), allowing the user to select a file. The program then sets the **`FileName`** property of the player (the Windows Media Player control object of type **`AxMediaPlayer`**) to the name of the file chosen by the user. The **`FileName`** property specifies the file that Windows Media Player currently is using. Lines 46–50 adjust the size of **`player`** and the application to reflect the size of the media contained in the file.

The event handler that executes when the user selects **Exit** from the **File** menu (lines 54–59) simply calls **`Application.Exit`** to terminate the application. The event handler that executes when the user chooses **About Windows Media Player** from the **About** menu (lines 61–66) calls the `AboutBox` method of the player. `AboutBox` simply displays a preset message box containing information about Windows Media Player.

16.13 Microsoft Agent

Microsoft Agent is a technology used to add *interactive animated characters* to Windows applications or Web pages. Interactivity is the key function of Microsoft Agent technology: Microsoft Agent characters can speak and respond to user input via speech recognition and synthesis. Microsoft employs its Agent technology in applications such as Word, Excel and PowerPoint. Agents in these programs aid users in finding answers to questions and in understanding how the applications function.

The Microsoft Agent control provides programmers with access to four predefined characters—*Genie* (a genie), *Merlin* (a wizard), *Peedy* (a parrot) and *Robby* (a robot). Each character has a unique set of animations that programmers can use in their applications to illustrate different points and functions. For instance, the Peedy character-animation set includes different flying animations, which the programmer might use to move Peedy on the screen. Microsoft provides basic information on Agent technology at its Web site,

`www.microsoft.com/msagent`

Microsoft Agent technology enables users to interact with applications and Web pages through speech, the most natural form of human communication. When the user speaks into a microphone, the control uses a *speech recognition engine*, an application that translates vocal sound input from a microphone into language that the computer understands. The Microsoft Agent control also uses a *text-to-speech engine*, which generates characters' spoken responses. A text-to-speech engine is an application that translates typed words into audio sound that users hear through headphones or speakers connected to a computer. Microsoft provides speech recognition and text-to-speech engines for several languages at its Web site,

`www.microsoft.com/products/msagent/downloads.htm`

Programmers can even create their own animated characters with the help of the *Microsoft Agent Character Editor* and the *Microsoft Linguistic Sound Editing Tool*. These products are available free for download from

`www.microsoft.com/products/msagent/devdownloads.htm`

This section introduces the basic capabilities of the Microsoft Agent control. For complete details on downloading this control, visit

`www.microsoft.com/products/msagent/downloads.htm`

The following example, Peedy's Pizza Palace, was developed by Microsoft to illustrate the capabilities of the Microsoft Agent control. Peedy's Pizza Palace is an online pizza shop where users can place their orders via voice input. The Peedy character interacts with users by helping them choose toppings and then calculating the totals for their orders.

Readers can view this example at

`agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm`

To run this example, students must download the Peedy character file, a text-to-speech engine and a speech-recognition engine. When the page loads, the browser prompts for these downloads. Follow the directions provided by Microsoft to complete installation.

When the window opens, Peedy introduces himself (Fig. 16.28), and the words he speaks appear in a cartoon bubble above his head. Notice that Peedy's animations correspond to the words he speaks.

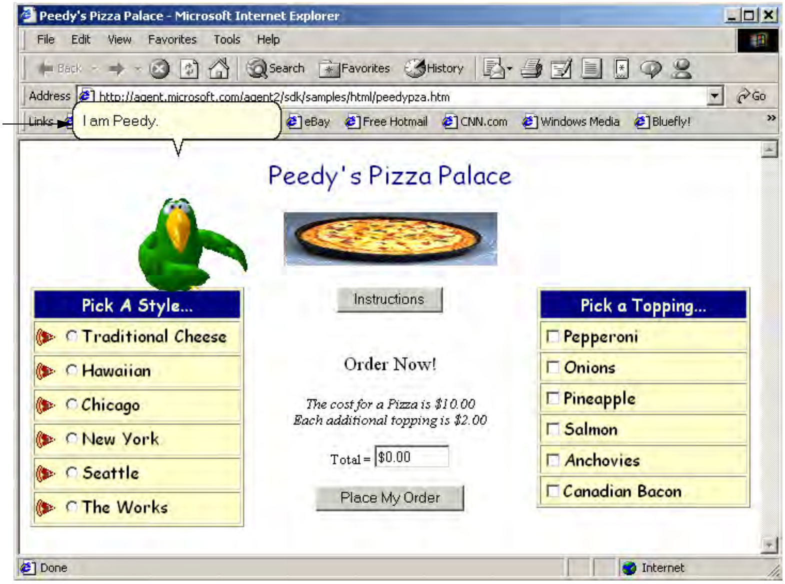
Programmers can synchronize character animations with speech output to illustrate a point or to convey a character's mood. For instance, Fig. 16.29 depicts Peedy's *Pleased* animation. The Peedy character-animation set includes eighty-five different animations, each of which is unique to the Peedy character.

Look-and-Feel Observation 16.1

Agent characters remain on top of all active windows while a Microsoft Agent application is running. Their motions are not limited to within the boundaries of the browser or application window.

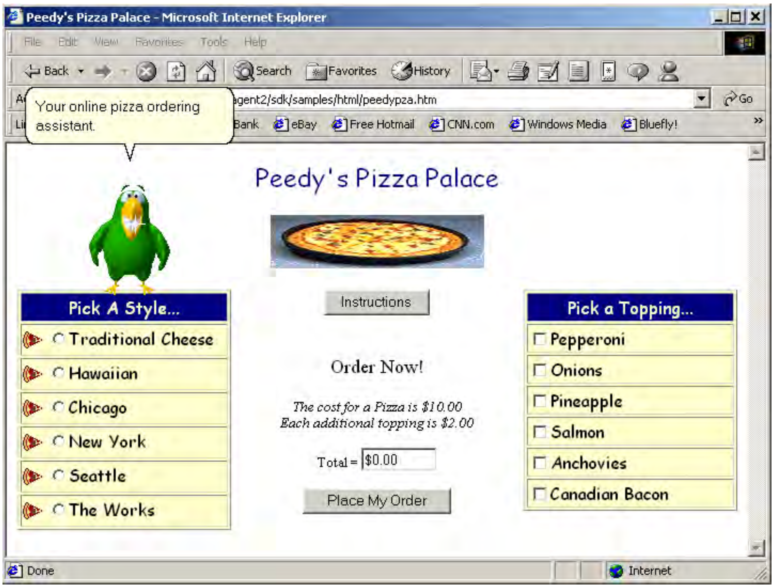
Peedy also responds to input from the keyboard and mouse. Figure 16.30 shows what happens when a user clicks Peedy with the mouse pointer. Peedy jumps up, ruffles his feathers and exclaims, "Hey that tickles!" or, "Be careful with that pointer!" Users can relocate Peedy on the screen by clicking and dragging him with the mouse. However, even when the user moves Peedy to a different part of the screen, he continues to perform his preset animations and location changes.

Bubble contains text equivalent to words Peedy speaks


Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm Go
Links eBay Free Hotmail CNN.com Windows Media Bluefly!
I am Peedy.
Peedy's Pizza Palace
Pick A Style...
○ Traditional Cheese
○ Hawaiian
○ Chicago
○ New York
○ Seattle
○ The Works
Instructions
Order Now!
Total = $0.00
Place My Order
Pick a Topping...
☐ Pepperoni
☐ Onions
☐ Pineapple
☐ Salmon
☐ Anchovies
☐ Canadian Bacon
Done
Internet

The cost for a Pizza is $10.00
Each additional topping is $2.00

Fig. 16.28 Peedy introducing himself when the window opens.

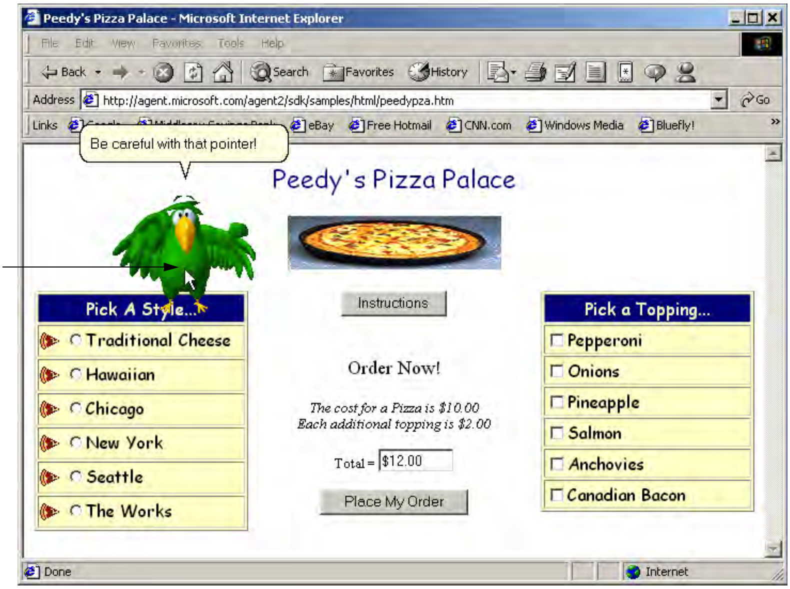

Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
agent2/sdk/samples/html/peedypza.htm
Go
Bank eBay Free Hotmail CNN.com Windows Media Bluefly!
Your online pizza ordering assistant.
Peedy's Pizza Palace
Instructions
Pick A Style...
○ Traditional Cheese
○ Hawaiian
○ Chicago
○ New York
○ Seattle
○ The Works
Pick a Topping...
☐ Pepperoni
☐ Onions
☐ Pineapple
☐ Salmon
☐ Anchovies
☐ Canadian Bacon
Place My Order
Done
Internet

Order Now!

The cost for a Pizza is $10.00
Each additional topping is $2.00

Total = $0.00

Fig. 16.29 Peedy's *Pleased* animation.


Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm
Go
Links eBay Free Hotmail CNN.com Windows Media Bluefly!
Be careful with that pointer!
Peedy's Pizza Palace
Instructions
Pick A Style...
○ Traditional Cheese
○ Hawaiian
○ Chicago
○ New York
○ Seattle
○ The Works
Pick a Topping...
☐ Pepperoni
☐ Onions
☐ Pineapple
☐ Salmon
☐ Anchovies
☐ Canadian Bacon
Place My Order
Done
Internet

Order Now!

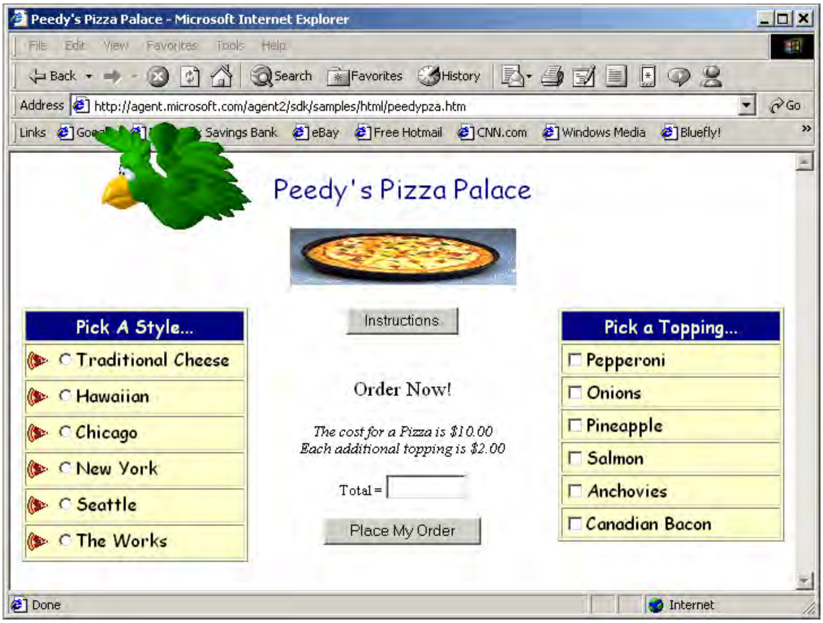
The cost for a Pizza is $10.00
Each additional topping is $2.00

Total = $12.00

Pointer clicking Peedy

Fig. 16.30 Peedy's reaction when he is clicked.

Many location changes involve animations. For instance, Peedy can hop from one screen location to another, or he can fly (Fig. 16.31).


Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm
Go
Links Savings Bank eBay Free Hotmail CNN.com Windows Media Bluefly!
Peedy's Pizza Palace
Pick A Style...
○ Traditional Cheese
○ Hawaiian
○ Chicago
○ New York
○ Seattle
○ The Works
Instructions
Order Now!
Total =
Place My Order
Pick a Topping...
☐ Pepperoni
☐ Onions
☐ Pineapple
☐ Salmon
☐ Anchovies
☐ Canadian Bacon
Done
Internet

The cost for a Pizza is $10.00
Each additional topping is $2.00

Fig. 16.31 Peedy flying animation.

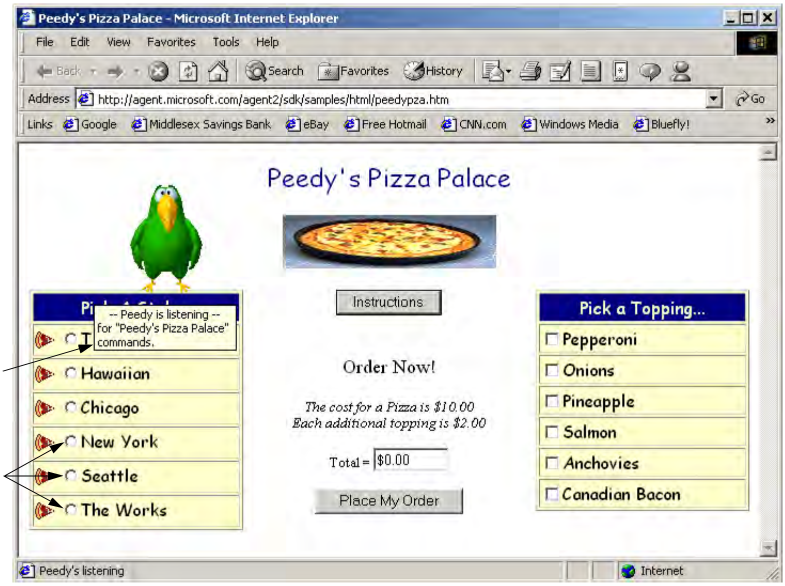
Once Peedy completes the ordering instructions, a text box appears beneath him indicating that he is listening for a voice command (Fig. 16.32). A user can enter the type of pizza to order either by speaking the style name into a microphone or by clicking the radio button corresponding to their choice.

If a user chooses speech input, a box appears below Peedy displaying the words that Peedy "heard" (i.e., the words translated to the program by the speech-recognition engine). Once he recognizes the user input, Peedy gives the user a description of the selected pizza. Figure 16.33 shows what happens when the user chooses **Seattle** as the pizza style.

Peedy then asks the user to choose additional toppings. Again, the user can either speak or use the mouse to make a selection. Check boxes corresponding to toppings that come with the selected pizza style are checked for the user. Figure 16.34 shows what happens when a user chooses anchovies as an additional topping. Peedy makes a wisecrack about the user's choice.

The user can submit the order either by pressing the **Place My Order** button or by speaking "Place order" into the microphone. Peedy recounts the order while writing down the order items on his notepad (Fig. 16.35). He then calculates the figures on his calculator and reports the total to the user (Fig. 16.36).

The following example (Fig. 16.37) demonstrates how to build a simple application with the Microsoft Agent control. This application contains two drop-down lists from which the user can choose an Agent character and a character animation. When the user chooses from these lists, the chosen character appears and performs the chosen animation. The application uses speech recognition and synthesis to control the character animations and speech: Users can tell the character which animation to perform by pressing the Scroll Lock key and then speaking the animation name into a microphone.



Peedy's Pizza Palace - Microsoft Internet Explorer

Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm

-- Peedy is listening --
for "Peedy's Pizza Palace"
commands.

Order Now!

The cost for a Pizza is $10.00
Each additional topping is $2.00

Total = $0.00

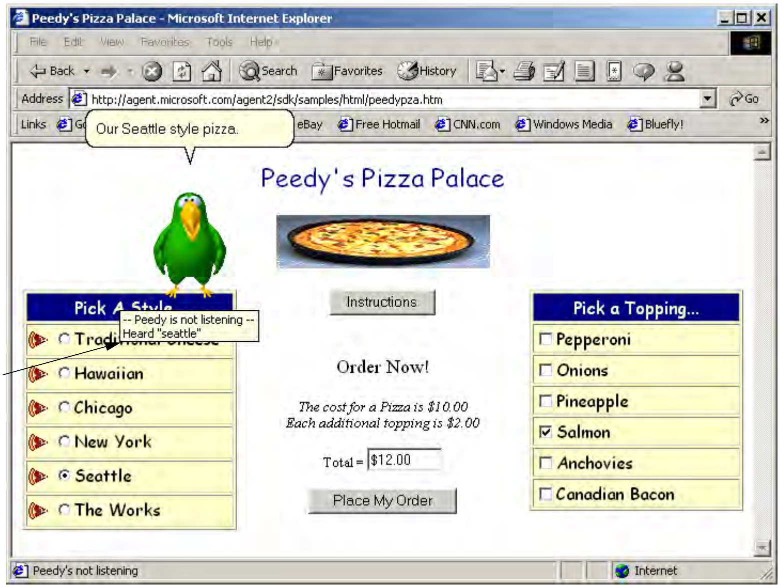
Peedy's listening

File Edit View Favorites Tools Help
Back Search Favorites History
Go
Links Google Middlesex Savings Bank eBay Free Hotmail CNN.com Windows Media Bluefly!
Peedy's Pizza Palace
Pick A Style
Hawaiian
Chicago
New York
Seattle
The Works
Instructions
Place My Order
Pick a Topping...
☐ Pepperoni
☐ Onions
☐ Pineapple
☐ Salmon
☐ Anchovies
☐ Canadian Bacon
Internet

Text box indicates that Peedy is waiting for user input

Radio buttons corresponding to different pizza styles

Fig. 16.32 Peedy waiting for speech input.



Peedy's Pizza Palace - Microsoft Internet Explorer

Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm

Our Seattle style pizza.

-- Peedy is not listening --
Heard "seattle"

Order Now!

The cost for a Pizza is $10.00
Each additional topping is $2.00

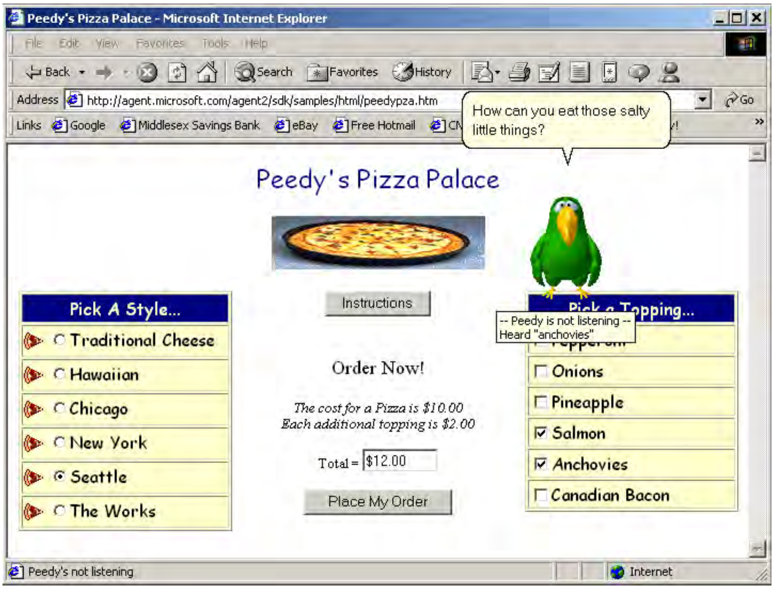
Total = $12.00

Peedy's not listening

File Edit View Favorites Tools Help
Back Search Favorites History
Go
Links eBay Free Hotmail CNN.com Windows Media Bluefly!
Peedy's Pizza Palace
Pick A Style
Traditional Cheese
Hawaiian
Chicago
New York
● Seattle
The Works
Instructions
Place My Order
Pick a Topping...
☐ Pepperoni
☐ Onions
☐ Pineapple
☑ Salmon
☐ Anchovies
☐ Canadian Bacon
Internet

Text box indicates recognized speech

Fig. 16.33 Peedy repeating the user's request for Seattle-style pizza.


Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm Go
Links Google Middlesex Savings Bank eBay Free Hotmail
Instructions
Place My Order
Internet

How can you eat those salty little things?

-- Peedy is not listening --
Heard "anchovies"

Peedy's Pizza Palace

Pick A Style...
- ○ Traditional Cheese
- ○ Hawaiian
- ○ Chicago
- ○ New York
- ● Seattle
- ○ The Works

Order Now!

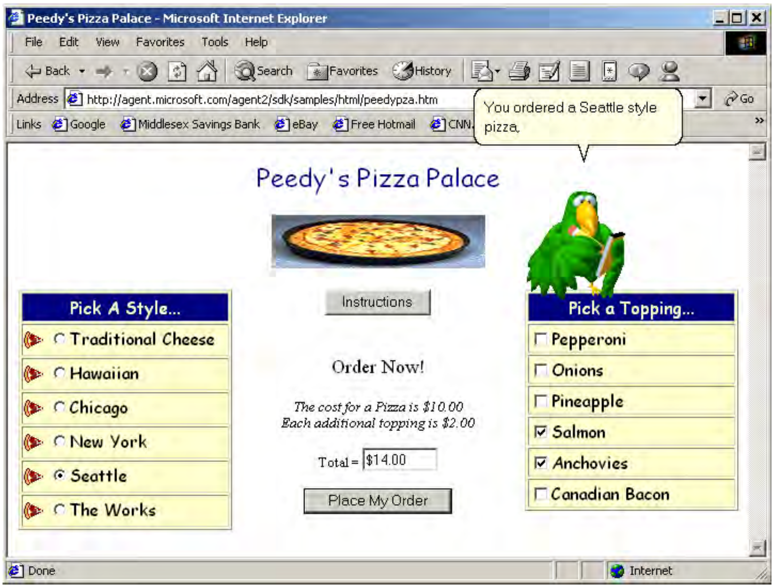
The cost for a Pizza is $10.00
Each additional topping is $2.00

Total = $12.00

Pick a Topping...
- ☐ Onions
- ☐ Pineapple
- ☑ Salmon
- ☑ Anchovies
- ☐ Canadian Bacon

Peedy's not listening

Fig. 16.34 Peedy repeating the user's request for anchovies as an additional topping.


Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm Go
Links Google Middlesex Savings Bank eBay Free Hotmail CNN
Instructions
Place My Order
Internet

You ordered a Seattle style pizza.

Peedy's Pizza Palace

Pick A Style...
- ○ Traditional Cheese
- ○ Hawaiian
- ○ Chicago
- ○ New York
- ● Seattle
- ○ The Works

Order Now!

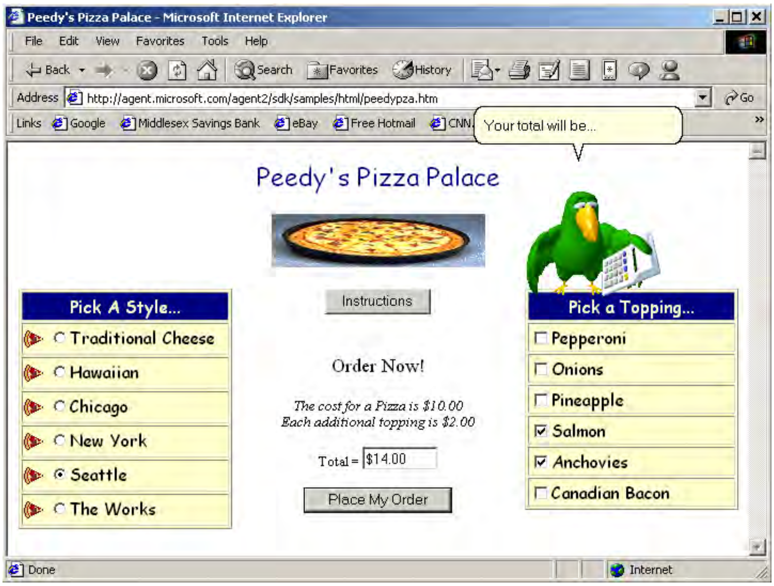
The cost for a Pizza is $10.00
Each additional topping is $2.00

Total = $14.00

Pick a Topping...
- ☐ Pepperoni
- ☐ Onions
- ☐ Pineapple
- ☑ Salmon
- ☑ Anchovies
- ☐ Canadian Bacon

Done

Fig. 16.35 Peedy recounting the order.


Peedy's Pizza Palace - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://agent.microsoft.com/agent2/sdk/samples/html/peedypza.htm
Go
Links Google Middlesex Savings Bank eBay Free Hotmail CNN
Your total will be...
Instructions
Place My Order
Done
Internet

Peedy's Pizza Palace

Pick A Style...

○ Traditional Cheese
○ Hawaiian
○ Chicago
○ New York
● Seattle
○ The Works

Order Now!

The cost for a Pizza is $10.00
Each additional topping is $2.00

Total = $14.00

Pick a Topping...

☐ Pepperoni
☐ Onions
☐ Pineapple
☑ Salmon
☑ Anchovies
☐ Canadian Bacon

Fig. 16.36 Peedy calculating the total.

The example also allows the user to switch to a new character by speaking its name and also creates a custom command, `MoveToMouse`. In addition, the characters also speak any text that a user enters into the text box. Before running this example, readers first must download and install the control, speech-recognition engine, text-to-speech engine and character definitions from the Microsoft Agent Web site listed previously.

To use the Microsoft Agent control, the programmer first must add it to the **Toolbox**. Begin by selecting **Customize Toolbox** from the **Tools** menu to display the **Customize Toolbox** dialog. In the dialog, scroll down and select the option **Microsoft Agent Control 2.0**. When this option is selected properly, a small check mark appears in the box to the left of the option. Then, click **OK** to dismiss the dialog. The icon for the Microsoft Agent control now should appear at the bottom of the **Toolbox**.

In addition to the Microsoft Agent object `mainAgent` (of type `AxAgent`) that manages all the characters, we also need an object (of type `IAgentCtlCharacter`) to represent the current character. We create this object, named `speaker`, in line 30.

```
1  // Fig. 16.28:  Agent.cs
2  // Demonstrates microsoft agent
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Data;
```

Fig. 16.37 Microsoft Agent demonstration. (Part 1 of 7.)

```
using System.IO;

public class Agent : System.Windows.Forms.Form
{
   // options
   private System.Windows.Forms.ComboBox actionsCombo;
   private System.Windows.Forms.ComboBox characterCombo;

   private System.Windows.Forms.Button speakButton;
   private System.Windows.Forms.GroupBox characterGroup;
   private AxAgentObjects.AxAgent mainAgent;

   // input
   private System.Windows.Forms.TextBox speechTextBox;
   private System.Windows.Forms.TextBox locationTextBox;

   private
      System.ComponentModel.Container components = null;

   // current agent object
   private AgentObjects.IAgentCtlCharacter speaker;

   [STAThread]
   static void Main()
   {
      Application.Run( new Agent() );
   }

   // Visual Studio .NET generated code

   // KeyDown event handler for locationTextBox
   private void locationTextBox_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      if ( e.KeyCode == Keys.Enter )
      {
         // set character location to text box value
         string location = locationTextBox.Text;

         // initialize the characters
         try
         {
            // load characters into agent object
            mainAgent.Characters.Load( "Genie",
               location + "Genie.acs" );

            mainAgent.Characters.Load( "Merlin",
               location + "Merlin.acs" );

            mainAgent.Characters.Load( "Peedy",
               location + "Peedy.acs" );

```

Fig. 16.37 Microsoft Agent demonstration. (Part 2 of 7.)

```
               mainAgent.Characters.Load( "Robby",
                  location + "Robby.acs" );

               // disable TextBox for entering the location
               // and enable other controls
               locationTextBox.Enabled = false;
               speechTextBox.Enabled = true;
               speakButton.Enabled = true;
               characterCombo.Enabled = true;
               actionsCombo.Enabled = true;

               // set current character to Genie and show him
               speaker = mainAgent.Characters[ "Genie" ];

               // obtain an animation name list
               GetAnimationNames();
               speaker.Show( 0 );
            }
            catch( FileNotFoundException )
            {
               MessageBox.Show( "Invalid character location",
                  "Error", MessageBoxButtons.OK,
                  MessageBoxIcon.Error );
            }
         }

      } // end method locationTextBox_KeyDown

      private void speakButton_Click(
         object sender, System.EventArgs e )
      {
         // if textbox is empty, have the character ask
         // user to type the words into textbox, otherwise
         // have character say the words in textbox
         if ( speechTextBox.Text == "" )
            speaker.Speak(
               "Please, type the words you want me to speak",
               "" );
         else
            speaker.Speak( speechTextBox.Text, "" );

      } // end method speakButton_Click

      // click event for agent
      private void mainAgent_ClickEvent( object sender,
         AxAgentObjects._AgentEvents_ClickEvent e )
      {
         speaker.Play( "Confused" );
         speaker.Speak( "Why are you poking me?", "" );
         speaker.Play( "RestPose" );

      } // end method mainAgent_ClickEvent

```

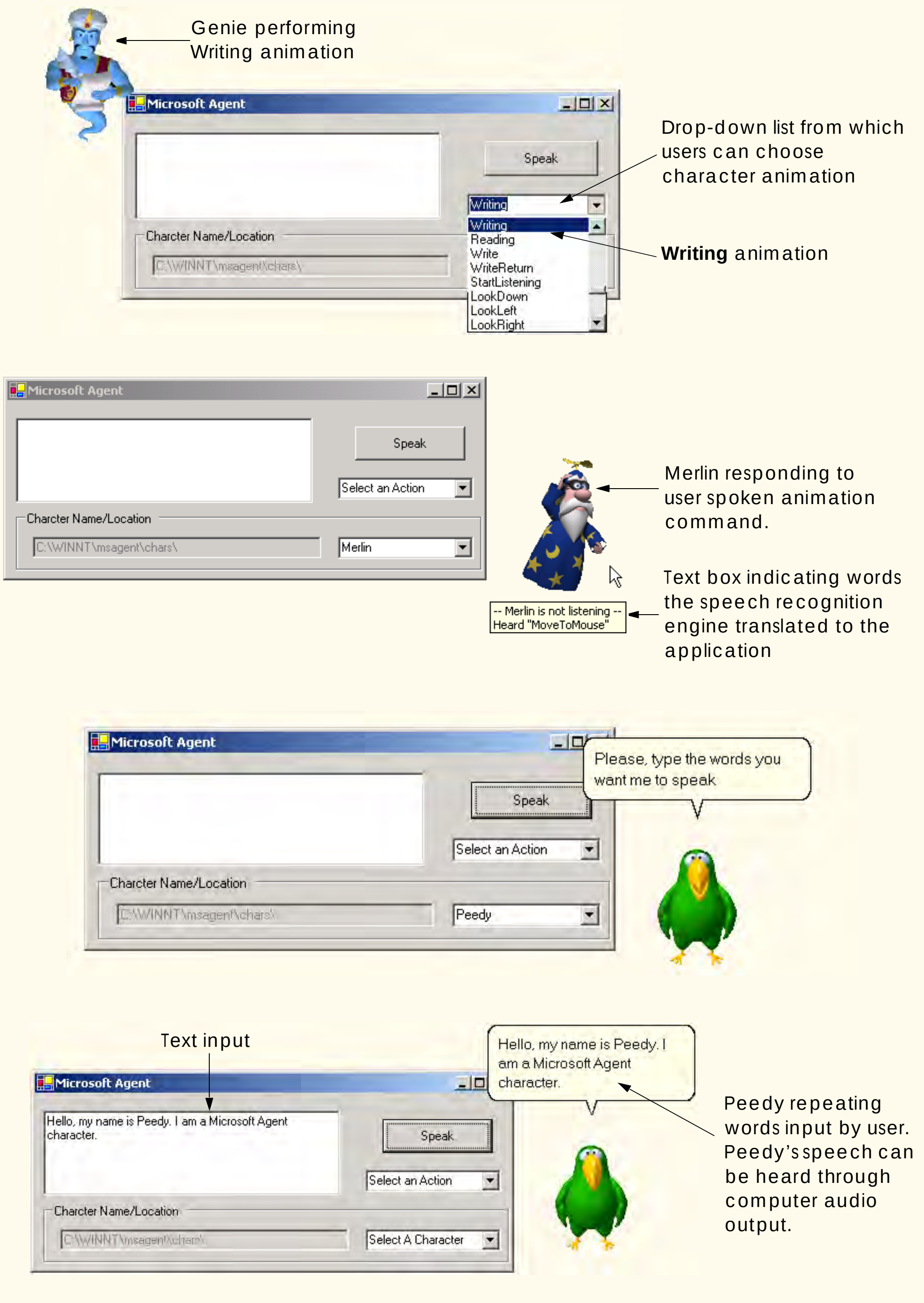
Fig. 16.37 Microsoft Agent demonstration. (Part 3 of 7.)

```
   // combobox changed event, switch active agent
   private void characterCombo_SelectedIndexChanged(
      object sender, System.EventArgs e )
   {
      ChangeCharacter( characterCombo.Text );

   } // end method characterCombo_SelectedIndexChanged

   private void ChangeCharacter( string name )
   {
      speaker.Hide( 0 );
      speaker = mainAgent.Characters[ name ];

      // regenerate animation name list
      GetAnimationNames();
      speaker.Show( 0 );

   } // end method ChangeCharacter

   // get animation names and store in arraylist
   private void GetAnimationNames()
   {
      // ensure thread safety
      lock( this )
      {

         // get animation names
         IEnumerator enumerator =
            mainAgent.Characters[
            speaker.Name ].AnimationNames.GetEnumerator();

         string voiceString;

         // clear actionsCombo
         actionsCombo.Items.Clear();
         speaker.Commands.RemoveAll();

         // copy enumeration to ArrayList
         while ( enumerator.MoveNext() )
         {
            //remove underscores in speech string
            voiceString = ( string )enumerator.Current;
            voiceString =
               voiceString.Replace( "_", "underscore" );

            actionsCombo.Items.Add( enumerator.Current );

            // add all animations as voice enabled commands
            speaker.Commands.Add(
               ( string )enumerator.Current,
               enumerator.Current,
               voiceString, true, false );
         }
```

Fig. 16.37 Microsoft Agent demonstration. (Part 4 of 7.)

```

         // add custom command
         speaker.Commands.Add(
            "MoveToMouse", "MoveToMouse",
            "MoveToMouse", true, true );
      }

   } // end method GetAnimationNames

   // user selects new action
   private void actionsCombo_SelectedIndexChanged(
      object sender, System.EventArgs e )
   {
      speaker.StopAll( "Play" );
      speaker.Play( actionsCombo.Text );
      speaker.Play( "RestPose" );

   } // end method actionsCombo_SelectedIndexChanged

   // handles agent commands
   private void mainAgent_Command( object sender,
      AxAgentObjects._AgentEvents_CommandEvent e )
   {
      // get userInput property
      AgentObjects.IAgentCtlUserInput command =
         ( AgentObjects.IAgentCtlUserInput )e.userInput;

      // change character if user speaks character name
      if ( command.Voice == "Peedy" ||
         command.Voice == "Robby" ||
         command.Voice == "Merlin" ||
         command.Voice == "Genie" )
      {
         ChangeCharacter( command.Voice );

         return;
      }

      // send agent to mouse
      if ( command.Voice == "MoveToMouse" )
      {
         speaker.MoveTo(
            Convert.ToInt16( Cursor.Position.X - 60 ),
            Convert.ToInt16( Cursor.Position.Y - 60 ), 5 );
         return;
      }

      // play new animation
      speaker.StopAll( "Play" );
      speaker.Play( command.Name );

   } // end method mainAgent_Command
} // end class Agent
```

Fig. 16.37 Microsoft Agent demonstration. (Part 5 of 7.)



Genie performing Writing animation

Drop-down list from which users can choose character animation

Writing animation

Merlin responding to user spoken animation command.

Text box indicating words the speech recognition engine translated to the application

-- Merlin is not listening --
Heard "MoveToMouse"

Please, type the words you want me to speak

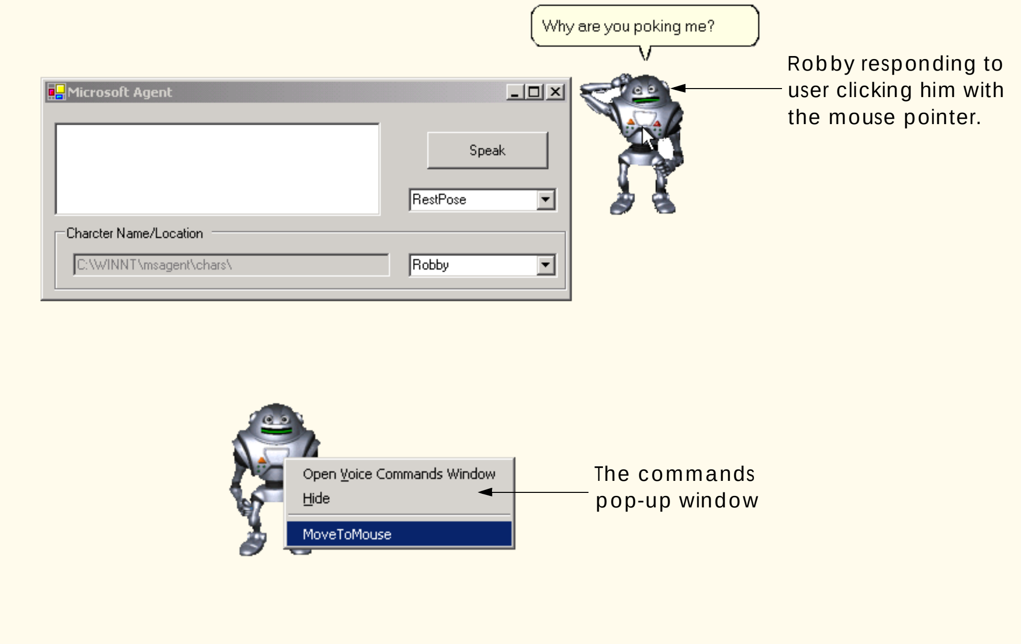
Text input

Hello, my name is Peedy. I am a Microsoft Agent character.

Peedy repeating words input by user. Peedy's speech can be heard through computer audio output.

Microsoft Agent
Speak
Writing
Writing
Reading
Write
WriteReturn
StartListening
LookDown
LookLeft
LookRight
Charcter Name/Location
C:\WINNT\msagent\chars\
Select an Action
Merlin
Peedy
Select A Character

Fig. 16.37 Microsoft Agent demonstration. (Part 6 of 7.)



Robby responding to user clicking him with the mouse pointer.

The commands pop-up window

Fig. 16.37 Microsoft Agent demonstration. (Part 7 of 7.)

When the program begins, the only enabled control is the **`locationTextBox`**. This text box contains the default location for the character files, but the user can change this location if the files are located elsewhere on the user's computer. Once the user presses *Enter* in the **`TextBox`**, event handler **`locationTextBox_KeyDown`** (lines 41–88) executes. Lines 53–63 load the character descriptions for the predefined animated characters. If the specified location of the characters is incorrect, or if any character is missing, a **`FileNotFoundException`** is thrown.

Lines 67–71 disable **`locationTextBox`** and enable the rest of the controls. Lines 74–78 set Genie as the default character, obtain all animation names via method **`GetAnimationNames`** and then call **`IAgentCtlCharacter`** method **`Show`** to display the character. We access characters through property **`Characters`** of **`mainAgent`**, which contains all characters that have been loaded. We use the indexer of the **`Characters`** property to specify the name of the character that we wish to load (Genie).

When a user clicks the character (i.e., pokes it with the mouse), event handler **`mainAgent_ClickEvent`** (lines 106–113) executes. First, **`speaker`** method **`Play`** plays an animation. This method accepts as an argument a **`string`** representing one of the predefined animations for the character (a list of animations for each character is available at the Microsoft Agent Web site; each character provides over 70 animations). In our example, the argument to **`Play`** is **`"Confused"`**—this animation is defined for all four characters, each of which expresses this emotion in a unique way. The character then speaks, **`"Why are you poking me?"`** via a call to method **`Speak`**. Finally, the **`RestPose`** animation is played, which returns the character to its neutral, resting pose.

The list of valid commands for a character is contained in property `Commands` of the `IAgentCtlCharacter` object (`speaker`, in this example). The commands for an Agent character can be viewed in the **Commands** pop-up window, which displays when the user right-clicks an Agent character (the last screenshot in Fig. 16.37). Method `Add` of property `Commands` adds a new command to the command list. Method `Add` takes three `string` arguments and two `bool` arguments. The first `string` argument identifies the name of the command, which we use to identify the command programmatically. The second `string` defines the command name as it appears in the **Commands** pop-up window. The third `string` defines the voice input that triggers the command. The first `bool` specifies whether the command is active, and the second `bool` indicates whether the command is visible in the **Commands** pop-up window. A command is triggered when the user selects the command from the **Commands** pop-up window or speaks the voice input into a microphone. Command logic is handled in the `Command` event of the `AxAgent` control (`mainAgent`, in this example). In addition, Agent defines several global commands that have predefined functions (for example, speaking a character name causes that character to appear).

Method `GetAnimationNames` (lines 135–175) fills the `actionsCombo` `ComboBox` with the current character's animation listing and defines the valid commands that can be used with the character. The method contains a `lock` block to prevent errors resulting from rapid character changes. The method obtains the current character's animations as an enumerator (142–144), then clears the existing items in the `ComboBox` and character's `Commands` property. Lines 153–167 iterate through all items in the animation-name enumerator. For each animation, in line 156, we assign the animation name to `string voiceString`. Lines 157–158 remove any underscore characters (_) and replaces them with the `string "underscore"`; this changes the `string` so that a user can pronounce and employ it as a command activator. The `Add` method (lines 163–166) of the `Commands` property adds a new command to the current character. The `Add` method adds all animations as commands by providing the following arguments: The animation name as the new command's `name` and caption, and `voiceString` for the voice activation `string`. The method's `bool` arguments enable the command, but make it unavailable in the **Commands** pop-up window. Thus, the command can be activated only by voice input. Lines 170–172 create a new command, named `MoveToMouse`, which is visible in the **Commands** pop-up window.

After the `GetAnimationNames` method has been called, the user can select a value from the `actionsCombo` `ComboBox`. Event handler `actionsCombo.SelectedIndexChanged` stops any current animation and then displays the animation that the user selected from the `ComboBox`.

The user also can type text into the `TextBox` and click **Speak**. This causes event handler `speakButton_Click` (line 90–103) to call `speaker`'s method `Speak`, supplying as an argument the text in `speechTextBox`. If the user clicks **Speak** without providing text, the character speaks, `"Please, type the words you want me to speak"`.

At any point in the program, the user can choose to display a different character from the `ComboBox`. When this happens, the `SelectedIndexChanged` event handler for `characterCombo` (lines 116–121) executes. The event handler calls method `ChangeCharacter` (lines 123–132) with the text in the `characterCombo` `ComboBox` as an argument. Method `ChangeCharacter` calls the `Hide` method of `speaker` (line 125) to remove the current character from view. Line 126 assigns the newly selected character to **speaker**, line 129 generates the character's animation names and commands, and line 130 displays the character via a call to method **Show**.

Each time a user presses the *Scroll Lock* key and speaks into a microphone or selects a command from the **Commands** pop-up window, event handler **mainAgent_Command** is called. This method is passed an argument of type **AxAgentObjects._AgentEvents_CommandEvent**, which contains a single property, **userInput**. The **userInput** property returns an **Object** that can be converted to type **AgentObjects.IAgentCtlUserInput**. The **userInput** object is assigned to a **IAgentCtlUserInput** object **command**, which is used to identify the command and then take appropriate action. Lines 196–204 use method **ChangeCharacter** to change the current Agent character if the user speaks a character name. Microsoft Agent always will show a character when a user speaks its name; however, by controlling the character change, we can ensure that only one Agent character is displayed at a time. Lines 207–213 move the character to the current mouse location if the user invokes the **MoveToMouse** command. The Agent method **MoveTo** takes x- and y-coordinate arguments and moves the character to the specified screen position, applying appropriate movement animations. For all other commands, we **Play** the command name as an animation on line 217.

In this chapter, we explored various graphics capabilities of GDI+, including pens, brushes and images, and some multimedia capabilities of the .NET Framework Class Library. In the next chapter, we cover the reading, writing and accessing of sequential- and random-access files. We also explore several types of streams included in Visual Studio .NET.

SUMMARY

- A coordinate system is used to identify every possible point on the screen.
- The upper-left corner of a GUI component has coordinates (0, 0). A coordinate pair is composed of an *x*-coordinate (the horizontal coordinate) and a *y*-coordinate (the vertical coordinate).
- Coordinate units are measured in pixels. A pixel is the smallest unit of resolution on a display monitor.
- A graphics context represents a drawing surface on the screen. A **Graphics** object provides access to the graphics context of a control.
- An instance of the **Pen** class is used to draw lines.
- An instance of one of the classes that derive from abstract class **Brush** is used to draw solid shapes.
- The **Point** structure can be used to represent a point in a two-dimensional plane.
- **Graphics** objects contain methods for drawing, font manipulation, color manipulation and other graphics-related actions.
- Method **OnPaint** normally is called in response to an event, such as the uncovering of a window. This method, in turn, triggers a **Paint** event.
- Structure **Color** defines constants for manipulating colors in a C# program.
- **Color** properties **R**, **G** and **B** return **int** values from 0 to 255, representing the amounts of red, green and blue, respectively, that exist in a **Color**. The larger the value, the greater the amount of that particular color.
- C# provides class **ColorDialog** to display a dialog that allows users to select colors.
- **Component** property **BackColor** (one of the many **Component** properties that can be called on most GUI components) changes the component's background color.

- Class **Font**'s constructors all take at least three arguments—the font name, the font size and the font style. The font name is any font currently supported by the system. The font style is a member of the **FontStyle** enumeration.
- Class **FontMetrics** defines several methods for obtaining font metrics.
- Class **Font** provides the **Bold**, **Italic**, **Strikeout** and **Underline** properties, which return **true** if the font is bold, italic, strikeout or underlined, respectively.
- Class **Font** provides the **Name** property, which returns a **string** representing the name of the font.
- Class **Font** provides the **Size** and **SizeInPoints** properties, which return the size of the font in design units and points, respectively.
- The **FontFamily** class provides information about such font metrics as the family's spacing and height.
- The **FontFamily** class provides the **GetCellAscent**, **GetCellDescent**, **GetEmHeight** and **GetLineSpacing** methods, which return the ascent of a font, the descent of a font, the font's height in points and the distance between two consecutive lines of text, respectively.
- Class **Graphics** provides methods **DrawLine**, **DrawRectangle**, **DrawEllipse**, **DrawArc**, **DrawLines**, **DrawPolygon** and **DrawPie**, which draw lines and shape outlines.
- Class **Graphics** provides methods **FillRectangle**, **FillEllipse**, **FillPolygon** and **FillPie**, which draw solid shapes.
- Classes **HatchBrush**, **LinearGradientBrush**, **PathGradientBrush** and **TextureBrush** all derive from class **Brush** and represent shape-filling styles.
- **Graphics** method **FromImage** retrieves the **Graphics** object associated with the image file that is its argument.
- The **DashStyle** and **DashCap** enumerations define the style of dashes and their ends, respectively.
- Class **GraphicsPath** represents a shape constructed from straight lines and curves.
- **GraphicsPath** method **AddLine** appends a line to the shape that is encapsulated by the object.
- **GraphicsPath** method **CloseFigure** completes the shape that is represented by the **GraphicsPath** object.
- Class **Image** is used to manipulate images.
- Class **Image** provides method **FromFile** to retrieve an image stored on disk and load it into an instance of class **Image**.
- **Graphics** method **Clear** paints the entire **Control** with the color that the programmer provides as an argument.
- **Graphics** method **DrawImage** draws the specified **Image** on the **Control**.
- Using Visual Studio .NET and C#, programmers can create applications that use components such as Windows Media Player and Microsoft Agent.
- The Windows Media Player allows programmers to create applications that can play multimedia files.
- Microsoft Agent is a technology that allows programmers to include interactive animated characters in their applications.

TERMINOLOGY

A property of structure **Color**
AboutBox method of class **AxMediaPlayer**
Add method of class **ArrayList**
AddLine method of class **GraphicsPath**

animated characters
animating a series of images
animation
arc angle
arc method
ARGB values
ArrayList class
ascent of a font
audio-video interleave (AVI)
AxAgent class
AxMediaPlayer class
B property of structure **Color**
bandwidth
Bitmap class
Black static property of structure **Color**
Blue static property of structure **Color**
Bold member of enumeration **FontStyle**
Bold property of class **Font**
bounding rectangle for an oval
Brush class
Characters property of class **AxAgent**
closed polygon
CloseFigure method of class **GraphicsPath**
color constants
color manipulation
Color methods and properties
Color property of class **ColorDialog**
Color structure
ColorDialog class
complex curve
connected lines
coordinate system
coordinates (0, 0)
curve
customizing the **Toolbox**
Cyan static property of structure **Color**
DarkBlue static property of structure **Color**
DarkGray static property of structure **Color**
Dash member of enumeration **DashStyle**
DashCap enumeration
DashCap property of class **Pen**
dashed lines
DashStyle enumeration
DashStyle property of class **Pen**
default font
degree
descent of a font
Display member of enumeration **GraphicsUnit**
display monitor
Document member of enumeration **GraphicsUnit**
DrawArc method of class **Graphics**
DrawEllipse method of class **Graphics**
DrawLine method of class **Graphics**
DrawLines method of class **Graphics**
DrawPie method of class **Graphics**
DrawPolygon method of class **Graphics**
DrawRectangle method of class **Graphics**
DrawString method of class **Graphics**
event-driven process
FileName property of class **AxMediaPlayer**
Fill method of class **Graphics**
fill a shape with color
FillEllipse method of class **Graphics**
fill shape
FillPie method of class **Graphics**
FillPolygon method of class **Graphics**
FillRectangle method of class **Graphics**
FillRectangles method of class **Graphics**
five-pointed star
font
font ascent
Font class
font control
font descent
font height
font leading
font manipulation
font metrics
font name
font size
font style
FontFamily class
FontFamily property of class **Font**
FontStyle enumeration
ForwardDiagonal member of enumeration **LinearGradientMode**
FromArgb method of structure **Color**
FromImage method of class **Graphics**
FromName method
G property of structure **Color**
GDI+
general path
Genie **Microsoft Agent** character
GetCellAscent method of class **FontFamily**

GetCellDescent method of class **FontFamily**
GetEmHeight method of class **FontFamily**
GetLineSpacing method of class **FontFamily**
graphics
Graphics class
graphics context
GraphicsPath class
GraphicsUnit
Gray static property of structure **Color**
Green static property of structure **Color**
HatchBrush class
HatchStyle enumeration
Height property of class **Font**
horizontal coordinate
IAgentCtlCharacter interface
Inch member of enumeration **GraphicsUnit**
interactive animated character
Invalidate method of class **Control**
Italic member of enumeration **FontStyle**
Italic property of class **Font**
line
LinearGradientBrush class
LinearGradientMode enumeration
Magenta static property of structure **Color**
Merlin **Microsoft Agent** character
Microsoft Agent
Microsoft Agent Character Editor
Microsoft Linguistic Sound-Editing Tool
Microsoft Sans Serif font
Microsoft Serif font
MIDI
Millimeter member of enumeration **GraphicsUnit**
Motion Pictures Experts Group (MPEG)
multimedia
Musical Instrument Digital Interface (MIDI)
Name property of class **Font**
Name property of structure **Color**
negative arc angles
OnPaint method of class **Control**
Orange static property of structure **Color**
PaintEventArgs class
Panel class
PathGradientBrush class
pattern
Peedy **Microsoft Agent** character
Pen class
Pink static property of structure **Color**
pixel
Pixel member of enumeration **GraphicsUnit**
Play method of interface **IAgentCtlCharacter**
Point member of enumeration **GraphicsUnit**
Point structure
positive and negative arc angles
R property of structure **Color**
rectangle
Rectangle structure
Red static property of structure **Color**
Regular member of enumeration **FontStyle**
resolution
RGB values
Robby the Robot **Microsoft Agent** character
RotateTransform method of class **Graphics**
Round member of enumeration **DashCap**
sector
Show method of interface **IAgentCtlCharacter**
Size property of class **Font**
style of a font
SizeInPoints property of class **Font**
solid arc
solid polygon
solid rectangle
SolidBrush class
starting angle
straight line
Strikeout member of enumeration **FontStyle**
Strikeout property of class **Font**
sweep
sweep counterclockwise
System.Drawing namespace
System.Drawing.Drawing2D namespace
TextureBrush class
thick line
thin line
three-dimensional application
Tick event of class **Timer**
Timer class
TranslateTransform method of class **Graphics**
two-dimensional shape
Underline member of enumeration **FontStyle**

`Underline` property of class `Font`
upper-left corner of a GUI component
vertical coordinate
WAV
`White` `static` property of structure `Color`
Windows Media Player
Windows wave file format (WAV)
x-axis
x-coordinate
y-axis
y-coordinate
yellow
`Yellow` `static` property of structure `Color`

SELF-REVIEW EXERCISES

16.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) A `Font` object's size can be changed by setting its `Size` property.
b) In the C# coordinate system, *x*-values increase from left to right.
c) Method `FillPolygon` draws a solid polygon with a specified `Brush`.
d) Method `DrawArc` allows negative angles.
e) `Font` property `Size` returns the size of the current font in centimeters.
f) Pixel coordinate (0, 0) is located at the exact center of the monitor.
g) A `HatchBrush` is used to draw lines.
h) A `Color` is defined by its alpha, red, green and violet content.
i) Every `Control` has an associated `Graphics` object.
j) Method `OnPaint` is inherited by every `Form`.

16.2 Fill in the blanks in each of the following statements:

a) Class __________ is used to draw lines of various colors and thicknesses.
b) Classes__________ and __________ define the fill for a shape in such a way that the fill gradually changes from one color to another.
c) The __________ method of class `Graphics` draws a line between two points.
d) ARGB is short for __________, __________, __________ and __________.
e) Font sizes usually are measured in units called __________.
f) Class__________ fills a shape using a pattern drawn in a `Bitmap`.
g) __________ __________ __________ allows an application to play multimedia files.
h) Class __________ defines a path consisting of lines and curves.
i) C#'s drawing capabilities are part of the namespaces __________ and __________.
j) Method __________ loads an image from a disk into an `Image` object.

ANSWERS TO SELF-REVIEW EXERCISES

16.1 a) False. `Size` is a read-only property. b) True. c) True. d) True. e) False. It returns the size of the current `Font` in design units. f) False. The coordinate (0,0) corresponds to the upper-left corner of a GUI component on which drawing occurs. g) False. A `Pen` is used to draw lines, a `HatchBrush` fills a shape with a hatch pattern. h) False. A color is defined by its alpha, red, green and blue content. i) True. j) True.

16.2 a) `Pen`. b) `LinearGradientBrush`, `PathGradientBrush`. c) `DrawLine`. d) alpha, red, green, blue. e) points. f) `TextureBrush`. g) Windows Media Player h) `GraphicsPath` i) `System.Drawing`, `System.Drawing.Drawing2D`. j) `FromFile`.

EXERCISES

16.3 Write a program that draws eight concentric circles. The circles should be separated from one another by 10 pixels. Use the `DrawArc` method.

16.4 Write a program that draws 100 lines with random lengths, positions, thicknesses and colors.

16.5 Write a program that draws a tetrahedron (a pyramid). Use class **`GraphicsPath`** and method **`DrawPath`**.

16.6 Write a program that allows the user to draw "free-hand" images with the mouse in a **`PictureBox`**. Allow the user to change the drawing color and width of the pen. Provide a button that allows the user to clear the **`PictureBox`**.

16.7 Write a program that repeatedly flashes an image on the screen. Do this by interspersing the image with a plain background-color image.

16.8 If you want to emphasize an image, you might place a row of simulated light bulbs around the image. Write a program by which an image is emphasized this way. You can let the light bulbs flash in unison or you can let them fire on and off in sequence, one after another.

16.9 (*Eight Queens*) A puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other (i.e., so that no two queens are in the same row, in the same column or along the same diagonal)?

Create a GUI that allows the user to drag-and-drop each queen on the board. Use the graphical features of Fig. 16.26. Provide eight queen images to the right of the board (Fig. 16.38), which the user can drag-and-drop onto the board. When a queen is dropped on the board, its corresponding image to the right should not be visible. If a queen is in conflict with another queen when placed on the board, display a message box and remove that queen from the board.

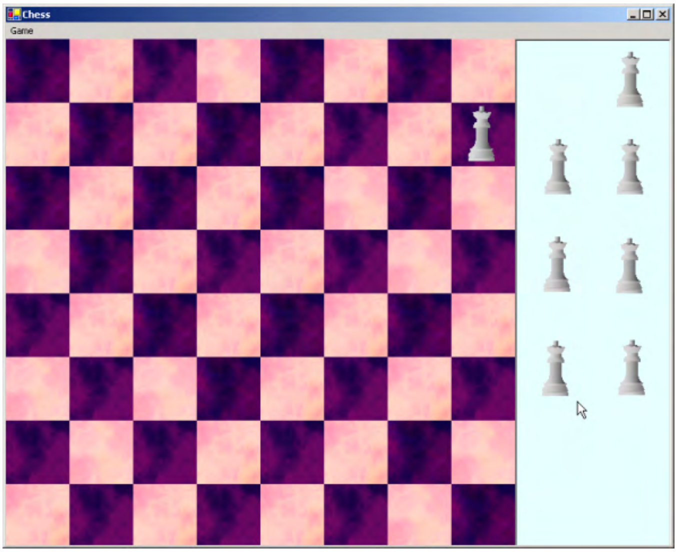


Fig. 16.38 GUI for Eight Queens exercise.

17

Files and Streams

Objectives

- To be able to create, read, write and update files.
- To understand the C# streams class hierarchy.
- To be able to use classes **`File`** and **`Directory`**.
- To be able to use the **`FileStream`** and **`BinaryFormatter`** classes to read objects from, and write objects to, files.
- To become familiar with sequential-access and random-access file processing.

I can only assume that a "Do Not File" document is filed in a "Do Not File" file.
Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975

Consciousness … does not appear to itself chopped up in bits. … A "river" or a "stream" are the metaphors by which it is most naturally described.
William James

I read part of it all the way through.
Samuel Goldwyn

Outline

17.1 Introduction
17.2 Data Hierarchy
17.3 Files and Streams
17.4 Classes `File` and `Directory`
17.5 Creating a Sequential-Access File
17.6 Reading Data from a Sequential-Access File
17.7 Random-Access Files
17.8 Creating a Random-Access File
17.9 Writing Data Randomly to a Random-Access File
17.10 Reading Data Sequentially from a Random-Access File
17.11 Case Study: A Transaction-Processing Program

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

17.1 Introduction

Variables and arrays offer only temporary storage of data—the data are lost when an object is garbage collected or when the program terminates. By contrast, *files* are used for long-term storage of large amounts of data and can retain data even after the program that created the data terminates. Data maintained in files often are called *persistent data*. Computers can store files on *secondary storage devices*, such as magnetic disks, optical disks and magnetic tapes. In this chapter, we explain how to create, update and process data files in C# programs. We consider both "sequential-access" files and "random-access" files, indicating the kinds of applications for which each is best suited. We have two goals in this chapter: To introduce the sequential-access and random-access file-processing paradigms and to provide the reader with sufficient stream-processing capabilities to support the networking features that we introduce in Chapter 22, Networking: Streams-Based Sockets and Datagrams.

File processing is one of a programming language's most important capabilities, because it enables a language to support commercial applications that typically process massive amounts of persistent data. This chapter discusses C#'s powerful and abundant file-processing and stream-input/output features.

17.2 Data Hierarchy

Ultimately, all data items processed by a computer are reduced to combinations of zeros and ones. This is because it is simple and economical to build electronic devices that can assume two stable states—`0` represents one state, and `1` represents the other. It is remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of `0`s and `1`s.

The smallest data items that computers support are called *bits* (short for "*binary digit*"—a digit that can assume one of two values). Each data item, or bit, can assume either the value `0` or the value `1`. Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from `1` to `0` or from `0` to `1`).

Programming with data in the low-level form of bits is cumbersome. It is preferable to program with data in forms such as *decimal digits* (i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9), *letters* (i.e., A through Z and a through z) and *special symbols* (i.e., $, @, %, &, *, (,), -, +, ", :, ?, / and many others). Digits, letters and special symbols are referred to as *characters*. The set of all characters used to write programs and represent data items on a particular computer is called that computer's *character set*. Because computers can process only `1`s and `0`s, every character in a computer's character set is represented as a pattern of `1`s and `0`s. *Bytes* are composed of eight bits (characters in C# are *Unicode* characters, which are composed of 2 bytes). Programmers create programs and data items with characters; computers manipulate and process these characters as patterns of bits.

In the same way that characters are composed of bits, *fields* are composed of characters. A field is a group of characters that conveys some meaning. For example, a field consisting of uppercase and lowercase letters can represent a person's name.

The various kinds of data items processed by computers form a *data hierarchy* (Fig. 17.1) in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and up to larger data structures.

Typically, a *record* is composed of several fields. In a payroll system, for example, a record for a particular employee might include the following fields:

1. Employee identification number
2. Name
3. Address
4. Hourly pay rate
5. Number of exemptions claimed
6. Year-to-date earnings
7. Amount of taxes withheld

Thus, a record is a group of related fields. In the preceding example, each field is associated with the same employee. A *file* is a group of related records.[1] A company's payroll file normally contains one record for each employee. Thus, a payroll file for a small company might contain only 22 records, whereas a payroll file for a large company might contain 100,000 records. It is not unusual for a company to have many files, some containing millions, billions or even trillions of bits of information.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a unique *record key*. A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all other records. In the payroll record described previously, the employee identification number normally would be chosen as the record key.

---

[1] More generally, a file can contain arbitrary data in arbitrary formats. In some operating systems, a file is viewed as nothing more than a collection of bytes. In such an operating system, any organization of the bytes in a file (such as organizing the data into records) is a view created by the applications programmer.

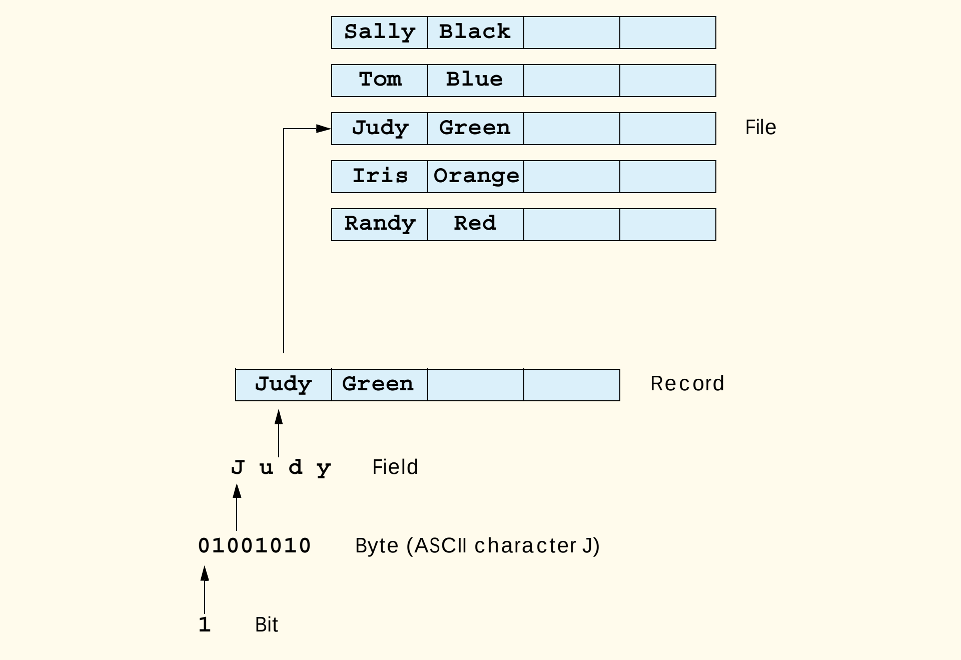


Fig. 17.1 Data hierarchy.

There are many ways of organizing records in a file. The most common type of organization is called a *sequential file*, in which records typically are stored in order by the record-key field. In a payroll file, records usually are placed in order by employee-identification numbers. The first employee record in the file contains the lowest employee-identification number, and subsequent records contain increasingly higher employee-identification numbers.

Most businesses use many different files to store data. For example, a company might have payroll files, accounts receivable files (listing money due from clients), accounts payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types of files. Sometimes, a group of related files is called a *database*. A collection of programs designed to create and manage databases is called a *database management system* (DBMS). We discuss databases in detail in Chapter 19, Databases, SQL and ADO.NET.

17.3 Files and Streams

C# views each file as a sequential *stream* of bytes (Fig. 17.2). Each file ends either with an *end-of-file marker* or at a specific byte number that is recorded in a system-maintained administrative data structure. When a file is *opened*, C# creates an object, then associates a stream with that object. The runtime environment creates three stream objects upon program execution, which are accessible via properties **`Console.Out`**, **`Console.In`** and **`Console.Error`**, respectively. These objects facilitate communication between a program and a particular file or device. Property **`Console.In`** returns the *standard input* *stream object*, which enables a program to input data from the keyboard. Property **`Console.Out`** returns the *standard output stream object*, which enables a program to output data to the screen. Property **`Console.Error`** returns the *standard error stream object*, which enables a program to output error messages to the screen. We have been using **`Console.Out`** and **`Console.In`** in our console applications—**`Console`** methods **`Write`** and **`WriteLine`** use **`Console.Out`** to perform output, and methods **`Read`** and **`ReadLine`** use **`Console.In`** to perform input.

To perform file processing in C#, namespace **`System.IO`** must be referenced. This namespace includes definitions for stream classes such as ***`StreamReader`*** (for text input from a file), ***`StreamWriter`*** (for text output to a file) and **`FileStream`** (for both input from and output to a file). Files are opened by creating objects of these stream classes, which inherit from **`abstract`** classes ***`TextReader`***, ***`TextWriter`*** and ***`Stream`***, respectively. Actually, **`Console.In`** and **`Console.Out`** are properties of class **`TextReader`** and **`TextWriter`**, respectively.

C# provides class ***`BinaryFormatter`***, which is used in conjunction with a **`Stream`** object to perform input and output of objects. *Serialization* involves converting an object into a format that can be written to a file without losing any of that object's data. *Deserialization* consists of reading this format from a file and reconstructing the original object from it. A **`BinaryFormatter`** can serialize objects to, and deserialize objects from, a specified **`Stream`**.

Class ***`System.IO.Stream`*** provides functionality for representing streams as bytes. This class is **`abstract`**, so objects of this class cannot be instantiated. Classes ***`FileStream`***, ***`MemoryStream`*** and ***`BufferedStream`*** (all from namespace **`System.IO`**) inherit from class **`Stream`**. Later in the chapter, we use **`FileStream`** to read data to, and write data from, sequential-access and random-access files. Class **`MemoryStream`** enables the transferal of data directly to and from memory—this type of transfer is much faster than are other types of data transfer (e.g., to and from disk). Class **`BufferedStream`** uses *buffering* to transfer data to or from a stream. Buffering is an I/O-performance-enhancement technique in which each output operation is directed to a region in memory called a *buffer* that is large enough to hold the data from many output operations. Then, actual transfer to the output device is performed in one large *physical output operation* each time the buffer fills. The output operations directed to the output buffer in memory often are called *logical output operations*.

C# offers many classes for performing input and output. In this chapter, we use several key stream classes to implement a variety of file-processing programs that create, manipulate and destroy sequential-access files and random-access files. In Chapter 22, Networking: Streams-Based Sockets and Datagrams, we use stream classes extensively to implement networking applications.

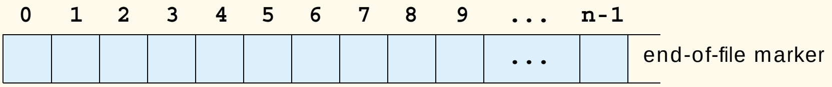


Fig. 17.2 C#'s view of an *n-byte* file.

17.4 Classes `File` and `Directory`

Information on computers is stored in files, which are organized in directories. Class ***`File`*** is provided for manipulating files, and class ***`Directory`*** is provided for manipulating directories. Class **`File`** cannot write to or read from files directly; we discuss methods for reading and writing files in subsequent sections.

Note that the **`\`** *separator character* separates directories and files in a path. On UNIX systems, the separator character is **`/`**. C# actually processes both characters as identical in a path name. This means that, if we specified the path **`c:\C_Sharp/README`**, which uses one of each separator character, C# still would process the file properly.

Figure 17.3 lists some methods contained in class **`File`** for manipulating and determining information about particular files. Class **`File`** contains only **`static`** methods—you cannot instantiate objects of type **`File`**. We use several of these methods in the example of Fig. 17.5.

Class **`Directory`** provides capabilities for manipulating directories. Figure 17.4 lists some methods that can be used for directory manipulation. We employ several of these methods in the example of Fig. 17.5

The ***`DirectoryInfo`*** object returned by method **`CreateDirectory`** contains information about a directory. Much of the information contained in this class also can be accessed via the methods of class **`Directory`**.

`static` Method	Description
`AppendText`	Returns a **`StreamWriter`** that appends to an existing file or creates a file if one does not exist.
`Copy`	Copies a file to a new file.
`Create`	Creates a file and returns its associated **`FileStream`**.
`CreateText`	Creates a text file and returns its associated **`StreamWriter`**.
`Delete`	Deletes the specified file.
`GetCreationTime`	Returns a **`DateTime`** object representing the time that the file was created.
`GetLastAccessTime`	Returns a **`DateTime`** object representing the time that the file was last accessed.
`GetLastWriteTime`	Returns a **`DateTime`** object representing the time that the file was last modified.
`Move`	Moves the specified file to a specified location.
`Open`	Returns a **`FileStream`** associated with the specified file and equipped with the specified read/write permissions.
`OpenRead`	Returns a read-only **`FileStream`** associated with the specified file.
`OpenText`	Returns a **`StreamReader`** associated with the specified file.
`OpenWrite`	Returns a read/write **`FileStream`** associated with the specified file.

Fig. 17.3 `File` class methods (partial list).

`static` Method	Description
`CreateDirectory`	Creates a directory and returns its associated **`DirectoryInfo`**.
`Delete`	Deletes the specified directory.
`Exists`	Returns **`true`** if the specified directory exists; otherwise, it returns **`false`**.
`GetLastWriteTime`	Returns a **`DateTime`** object representing the time that the directory was last modified.
`GetDirectories`	Returns a **`string`** array representing the names of the subdirectories in the specified directory.
`GetFiles`	Returns a **`string`** array representing the names of the files in the specified directory.
`GetCreationTime`	Returns a **`DateTime`** object representing the time that the directory was created.
`GetLastAccessTime`	Returns a **`DateTime`** object representing the time that the directory was last accessed.
`GetLastWriteTime`	Returns a **`DateTime`** object representing the time that items were last written to the directory.
`Move`	Moves the specified directory to a specified location.

Fig. 17.4 **`Directory`** class methods (partial list).

Class **`FileTestForm`** (Fig. 17.5) uses methods described in Fig. 17.3 and Fig. 17.4 to access file and directory information. This class contains **`TextBox inputTextBox`** (line 18), which enables the user to input a file or directory name. For each key that the user presses in the text box, the program calls method **`inputTextBox_KeyDown`** (lines 31–93). If the user presses the *Enter* key (line 35), this method displays either file or directory contents, depending on the text the user input in the **`TextBox`**. (Note that, if the user does not press the *Enter* key, this method returns without displaying any content.) Line 43 uses method **`Exists`** of class **`File`** to determine whether the user-specified text is a name of an existing file. If the user specifies an existing file, line 47 invokes **`private`** method **`GetInformation`** (lines 96–115), which calls methods **`GetCreationTime`** (line 103), **`GetLastWriteTime`** (line 107) and **`GetLastAccessTime`** (line 111) of class **`File`** to access file information. When method **`GetInformation`** returns, line 53 instantiates a **`StreamReader`** for reading text from the file. The **`StreamReader`** constructor takes as an argument a **`string`** containing the name of the file to open. Line 54 calls method **`ReadToEnd`** of the **`StreamReader`** to read the file content from the file, then displays the content.

```
1   // Fig 17.5: FileTest.cs
2   // Using classes File and Directory.
3
```

Fig. 17.5 Testing classes **`File`** and **`Directory`**. (Part 1 of 4.)

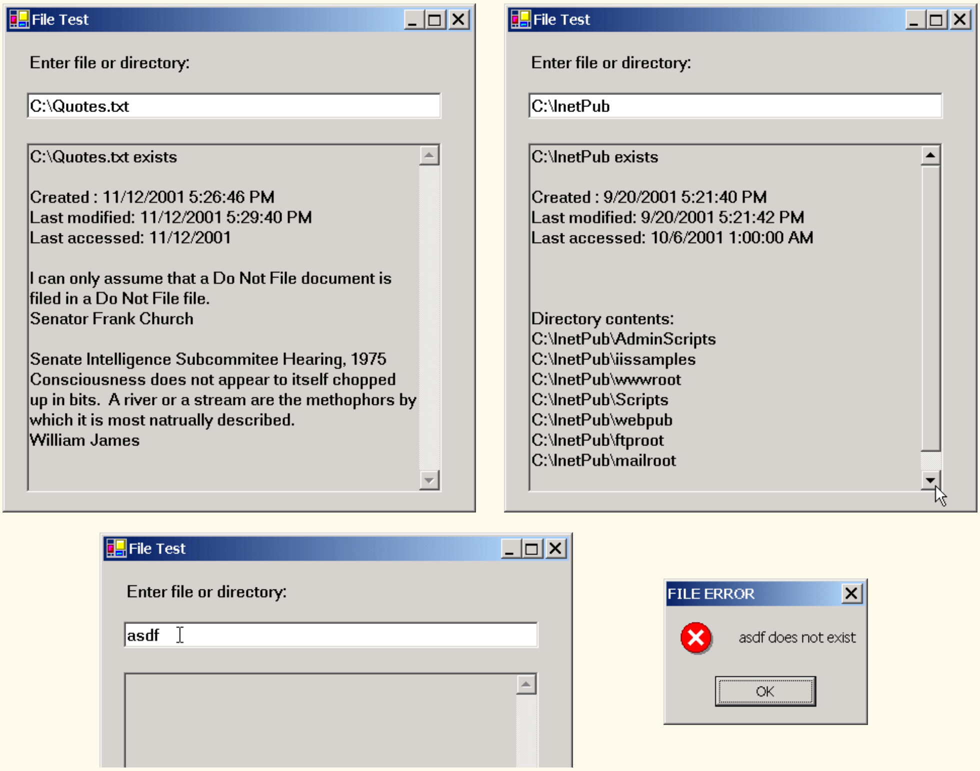
```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;

// displays contents of files and directories
public class FileTestForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label directionsLabel;

   private System.Windows.Forms.TextBox outputTextBox;
   private System.Windows.Forms.TextBox inputTextBox;

   private System.ComponentModel.Container components = null;

   [STAThread]
   static void Main()
   {
      Application.Run( new FileTestForm() );
   }

   // Visual Studio .NET generated code

   // invoked when user presses key
   private void inputTextBox_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      // determine whether user pressed Enter key
      if ( e.KeyCode == Keys.Enter )
      {
         string fileName; // name of file or directory

         // get user-specified file or directory
         fileName = inputTextBox.Text;

         // determine whether fileName is a file
         if ( File.Exists( fileName ) )
         {
            // get file's creation date,
            // modification date, etc.
            outputTextBox.Text = GetInformation( fileName );

            // display file contents through StreamReader
            try
            {
               // obtain reader and file contents
               StreamReader stream = new StreamReader( fileName );
               outputTextBox.Text += stream.ReadToEnd();
            }
```

Fig. 17.5 Testing classes `File` and `Directory`. (Part 2 of 4.)

```
            // handle exception if StreamReader is unavailable
            catch( IOException )
            {
               MessageBox.Show( "File Error", "File Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error );
            }
         }

         // determine whether fileName is a directory
         else if ( Directory.Exists( fileName ) )
         {
            // array for directories
            string[] directoryList;

            // get directory's creation date,
            // modification date, etc.
            outputTextBox.Text = GetInformation( fileName );

            // obtain file/directory list of specified directory
            directoryList = Directory.GetDirectories( fileName );

            outputTextBox.Text +=
               "\r\n\r\nDirectory contents:\r\n";

            // output directoryList contents
            for ( int i = 0; i < directoryList.Length; i++ )
               outputTextBox.Text += directoryList[ i ] + "\r\n";
         }
         else
         {
            // notify user that neither file nor directory exists
            MessageBox.Show( inputTextBox.Text +
               " does not exist", "File Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      } // end if

   } // end method inputTextBox_KeyDown

   // get information on file or directory
   private string GetInformation( string fileName )
   {
      // output that file or directory exists
      string information = fileName + " exists\r\n\r\n";

      // output when file or directory was created
      information += "Created: " +
         File.GetCreationTime( fileName ) + "\r\n";

      // output when file or directory was last modified
      information += "Last modified: " +
         File.GetLastWriteTime( fileName ) + "\r\n";

```

Fig. 17.5 Testing classes **`File`** and **`Directory`**. (Part 3 of 4.)

```
109          // output when file or directory was last accessed
110          information += "Last accessed: " +
111             File.GetLastAccessTime( fileName ) + "\r\n" + "\r\n";
112
113          return information;
114
115       } // end method GetInformation
116
117    } // end class FileTestForm
```



C:\Quotes.txt

```
C:\Quotes.txt exists

Created : 11/12/2001 5:26:46 PM
Last modified: 11/12/2001 5:29:40 PM
Last accessed: 11/12/2001

I can only assume that a Do Not File document is
filed in a Do Not File file.
Senator Frank Church

Senate Intelligence Subcommitee Hearing, 1975
Consciousness does not appear to itself chopped
up in bits.  A river or a stream are the methophors by
which it is most natrually described.
William James
```

C:\InetPub

```
C:\InetPub exists

Created : 9/20/2001 5:21:40 PM
Last modified: 9/20/2001 5:21:42 PM
Last accessed: 10/6/2001 1:00:00 AM



Directory contents:
C:\InetPub\AdminScripts
C:\InetPub\iissamples
C:\InetPub\wwwroot
C:\InetPub\Scripts
C:\InetPub\webpub
C:\InetPub\ftproot
C:\InetPub\mailroot
```

asdf

asdf does not exist

Fig. 17.5 Testing classes **`File`** and **`Directory`**. (Part 4 of 4.)

If line 43 determines that the user-specified text is not a file, line 65 determines whether it is a directory using method **`Exists`** of class **`Directory`**. If the user specified an existing directory, line 72 invokes method **`GetInformation`** to access the directory information. Line 75 calls method **`GetDirectories`** of class **`Directory`** to obtain a **`string`** array containing the names of subdirectories in the specified directory. Lines 81–82 display each element in the **`string`** array. Note that, if line 65 determines that the user-specified text is neither a file nor a directory, lines 87–89 notify the user (via a **`MessageBox`**) that the file or directory does not exist.

We now consider another example that uses C#'s file- and directory-manipulation capabilities. Class **`FileSearchForm`** (Fig. 17.6) uses classes **`File`** and **`Directory`** in conjunction with classes for performing regular expressions to report the number of files of each file type that exist in the specified directory path. The program also serves as a "clean-up" utility—when the program encounters a file that has the `.bak` extension (i.e., a backup file), the program displays a `MessageBox` asking whether that file should be removed, then responds appropriately to the user's input.

When the user presses the *Enter* key or clicks the **Search Directory** button, the program invokes method `searchButton_Click` (lines 52–92), which searches recursively through the directory path that the user provides. If the user inputs text in the `TextBox`, line 59 calls method `Exists` of class `Directory` to determine whether that text indicates a valid directory. If the user specifies an invalid directory, lines 70–71 notify the user of the error.

If the user specifies a valid directory, line 80 passes the directory name as an argument to `private` method `SearchDirectory` (lines 95–185). This method locates files that match the regular expression defined in lines 103–104, which matches any sequence of numbers or letters followed by a period and one or more letters. Notice the substring of format `(?<extension>`*regular-expression*`)` in the argument to the `Regex` constructor (line 104). All `string`s with the substring *regular-expression* are tagged with the name `extension`. In this program, we assign to the variable `extension` any `string` matching one or more characters.

Lines 115–116 call method `GetDirectories` of class `Directory` to retrieve the names of all subdirectories that belong to the current directory. Line 119 calls method `GetFiles` of class `Directory` to store in `string` array `fileArray` the names of files in the current directory. The `foreach` loop in lines 122–170 searches for all files with extension `bak`; it then calls `SearchDirectory` recursively for each subdirectory in the current directory. Lines 125–126 eliminate the directory path, so the program can test only the file name when using the regular expression. Line 129 uses method `Match` of the `Regex` object to match the regular expression with the file name, then returns the result to object `matchResult` of type `Match`. If the match is successful, lines 133–134 use method `Result` of object `matchResult` to store the extension `string` from object `matchResult` in `fileExtension` (the `string` that will contain the current file's extension). If the match is unsuccessful, line 136 sets `fileExtension` to hold a value of `"[no extension]"`.

```
1   // Fig 17.6: FileSearch.cs
2   // Using regular expressions to determine file types.
3
4   using System;
5   using System.Drawing;
6   using System.Collections;
7   using System.ComponentModel;
8   using System.Windows.Forms;
9   using System.Data;
10  using System.IO;
11  using System.Text.RegularExpressions;
12  using System.Collections.Specialized;
13
```

Fig. 17.6 Regular expression used to determine file types. (Part 1 of 5.)

```
public class FileSearchForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label directionsLabel;
   private System.Windows.Forms.Label directoryLabel;

   private System.Windows.Forms.Button searchButton;

   private System.Windows.Forms.TextBox outputTextBox;
   private System.Windows.Forms.TextBox inputTextBox;

   private System.ComponentModel.Container components = null;

   string currentDirectory = Directory.GetCurrentDirectory();
   string[] directoryList; // subdirectories
   string[] fileArray;

   // store extensions found and number found
   NameValueCollection found = new NameValueCollection();

   [STAThread]
   static void Main()
   {
      Application.Run( new FileSearchForm() );
   }

   // Visual Studio .NET generated code

   // invoked when user types in text box
   private void inputTextBox_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      // determine whether user pressed Enter
      if ( e.KeyCode == Keys.Enter )
         searchButton_Click( sender, e );

   } // end method inputTextBox_KeyDown

   // invoked when user clicks "Search Directory" button
   private void searchButton_Click(
      object sender, System.EventArgs e )
   {
      // check for user input; default is current directory
      if ( inputTextBox.Text != "" )
      {
         // verify that user input is valid directory name
         if ( Directory.Exists( inputTextBox.Text ) )
         {
            currentDirectory = inputTextBox.Text;

            // reset input text box and update display
            directoryLabel.Text = "Current Directory:" +
               "\r\n" + currentDirectory;
         }
```

Fig. 17.6 Regular expression used to determine file types. (Part 2 of 5.)

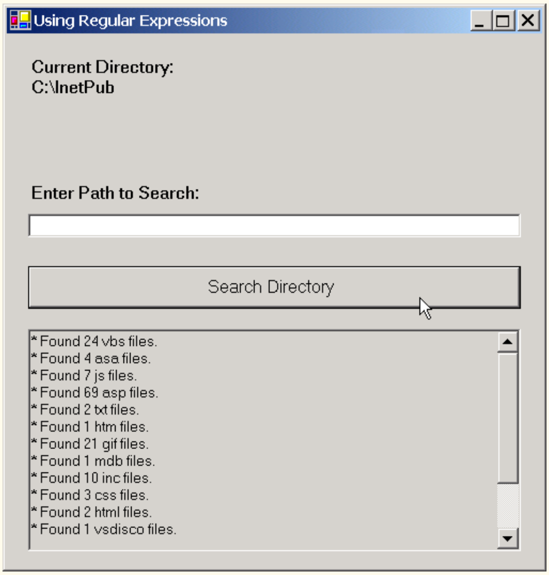
```
         else
         {
            // show error if user does not specify valid directory
            MessageBox.Show( "Invalid Directory", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }

      // clear text boxes
      inputTextBox.Clear();
      outputTextBox.Clear();

      // search directory
      SearchDirectory( currentDirectory );

      // summarize and print results
      foreach ( string current in found )
      {
         outputTextBox.Text += "* Found " +
            found[ current ] + " " + current + " files.\r\n";
      }

      // clear output for new search
      found.Clear();

   } // end method searchButton_Click

   // search directory using regular expression
   private void SearchDirectory( string currentDirectory )
   {
      // search directory
      try
      {
         string fileName = "";

         // regular expression for extensions matching pattern
         Regex regularExpression = new Regex(
            "[a-zA-Z0-9]+\\.(?<extension>\\w+)" );

         // stores regular-expression-match result
         Match matchResult;

         string fileExtension; // holds file extensions

         // number of files with given extension in directory
         int extensionCount;

         // get directories
         directoryList =
            Directory.GetDirectories( currentDirectory );

         // get list of files in current directory
         fileArray = Directory.GetFiles( currentDirectory );
```

Fig. 17.6 Regular expression used to determine file types. (Part 3 of 5.)

```

         // iterate through list of files
         foreach ( string myFile in fileArray )
         {
            // remove directory path from file name
            fileName = myFile.Substring(
               myFile.LastIndexOf( "\\" ) + 1 );

            // obtain result for regular-expression search
            matchResult = regularExpression.Match( fileName );

            // check for match
            if ( matchResult.Success )
               fileExtension =
                  matchResult.Result( "${extension}" );
            else
               fileExtension = "[no extension]";

            // store value from container
            if ( found[ fileExtension ] == null )
               found.Add( fileExtension, "1" );
            else
            {
               extensionCount = Int32.Parse(
                  found[ fileExtension ] ) + 1;

               found[ fileExtension ] = extensionCount.ToString();
            }

            // search for backup(.bak) files
            if ( fileExtension == "bak" )
            {
               // prompt user to delete (.bak) file
               DialogResult result =
                  MessageBox.Show( "Found backup file " +
                  fileName + ". Delete?", "Delete Backup",
                  MessageBoxButtons.YesNo,
                  MessageBoxIcon.Question );

               // delete file if user clicked 'yes'
               if ( result == DialogResult.Yes )
               {
                  File.Delete( myFile );

                  extensionCount =
                     Int32.Parse( found[ "bak" ] ) - 1;

                  found[ "bak" ] = extensionCount.ToString();
               }
            }
         }

```

Fig. 17.6 Regular expression used to determine file types. (Part 4 of 5.)

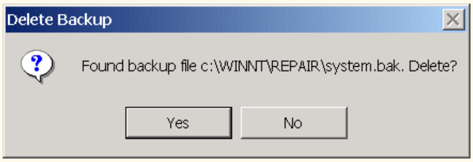
```
172            // recursive call to search files in subdirectory
173            foreach ( string myDirectory in directoryList )
174               SearchDirectory( myDirectory );
175         }
176
177         // handle exception if files have unauthorized access
178         catch( UnauthorizedAccessException )
179         {
180            MessageBox.Show( "Some files may not be visible" +
181               " due to permission settings", "Warning",
182               MessageBoxButtons.OK, MessageBoxIcon.Information );
183         }
184
185      } // end method SearchDirectory
186
187   } // end class FileSearchForm
```



Current Directory:
C:\InetPub

Enter Path to Search:

```
* Found 24 vbs files.
* Found 4 asa files.
* Found 7 js files.
* Found 69 asp files.
* Found 2 txt files.
* Found 1 htm files.
* Found 21 gif files.
* Found 1 mdb files.
* Found 10 inc files.
* Found 3 css files.
* Found 2 html files.
* Found 1 vsdisco files.
```



Found backup file c:\WINNT\REPAIR\system.bak. Delete?

Fig. 17.6 Regular expression used to determine file types. (Part 5 of 5.)

Class **`FileSearchForm`** uses an instance of class ***`NameValueCollection`*** (declared in line 31) to store each file-extension type and the number of files for each type. A **`NameValueCollection`** contains a collection of key/value pairs, each of which is a **`string`**, and provides method **`Add`** to add a key/value pair. The indexer for this pair can index according to the order that the items were added or according to the entry key. Line

139 uses **`NameValueCollection found`** to determine whether this is the first occurrence of the file extension. If so, line 140 adds that extension to **`found`** as a key with the value **`1`**. If the extension is in **`found`** already, lines 143–144 increment the value associated with the extension in **`found`** to indicate another occurrence of that file extension.

Line 150 determines whether **`fileExtension`** equals "**`bak`**"—i.e., whether the file is a backup file. If so, lines 153–157 prompt the user to indicate whether the file should be removed; if the user clicks **Yes** (line 160), lines 162–167 delete the file and decrement the value for the "**`bak`**" file type in **`found`**.

Lines 173–174 call method **`SearchDirectory`** for each subdirectory. Using recursion, we ensure that the program performs the same logic for finding **`bak`** files on each subdirectory. After each subdirectory has been checked for **`bak`** files, method **`SearchDirectory`** completes, and lines 83–87 display the results.

17.5 Creating a Sequential-Access File

C# imposes no structure on files. Thus, concepts like that of a "record" do not exist in C# files. This means that the programmer must structure files to meet the requirements of applications. In this example, we use text and special characters to organize our own concept of a "record."

The following examples demonstrate file processing in a bank-account maintenance application. These programs have similar user interfaces, so we created class **`BankUIForm`** (Fig. 17.7) to encapsulate a base-class GUI (see the screen capture in Fig. 17.7). Class **`BankUIForm`** contains four **`Label`**s (lines 15, 18, 21 and 24) and four **`TextBox`**es (lines 16, 19, 22 and 25). Methods **`ClearTextBoxes`** (lines 49–64), **`SetTextBoxValues`** (lines 67–91) and **`GetTextBoxValues`** (lines 94–110) clear, set the values of, and get the values of the text in the **`TextBox`**es, respectively.

To reuse class **`BankUIForm`**, we compile the GUI into a DLL library by creating a project of type **Windows Control Library** (the DLL we create is called **`BankLibrary`**). This library, as well as all the code in this book, can be found on the CD accompanying the book and at our Web site, **`www.deitel.com`**. However, students might need to change the reference to this library, as it most likely resides in a different location on their systems.

Figure 17.8 contains class **`Record`** that Fig. 17.9, Fig. 17.11 and Fig. 17.12 use for reading records from, and writing records to, a file sequentially. This class also belongs to the **`BankLibrary`** DLL, so it is located in the same project as is class **`BankUIForm`**.

The ***`Serializable`*** attribute (line 6) indicates to the compiler that objects of class **`Record`** can be *serialized*, or represented as sets of bytes—we can read and write these bytes to our streams. Objects that we wish to write to or read from a stream must include this attribute in their class definitions.

Class **`Record`** contains **`private`** data members **`account`**, **`firstName`**, **`lastName`** and **`balance`** (lines 9–12), which collectively represent all information necessary to store record data. The default constructor (lines 15–17) sets these members to their default (i.e., empty) values, and the overloaded constructor (lines 20–28) sets these members to specified parameter values. Class **`Record`** also provides properties **`Account`** (lines 31–43), **`FirstName`** (lines 46–58), **`LastName`** (lines 61–73) and **`Balance`** (lines 76–88) for accessing the account number, first name, last name and balance of each customer, respectively.

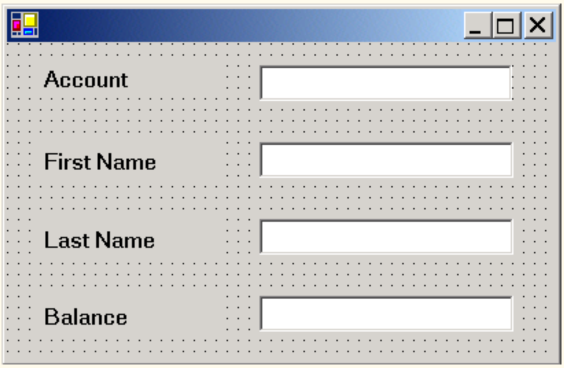
```
// Fig 17.7: BankUI.cs
// A reusable windows form for the examples in this chapter.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class BankUIForm : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;

   public System.Windows.Forms.Label accountLabel;
   public System.Windows.Forms.TextBox accountTextBox;

   public System.Windows.Forms.Label firstNameLabel;
   public System.Windows.Forms.TextBox firstNameTextBox;

   public System.Windows.Forms.Label lastNameLabel;
   public System.Windows.Forms.TextBox lastNameTextBox;

   public System.Windows.Forms.Label balanceLabel;
   public System.Windows.Forms.TextBox balanceTextBox;

   // number of TextBoxes on Form'
   protected int TextBoxCount = 4;

   // enumeration constants specify TextBox indices
   public enum TextBoxIndices
   {
      ACCOUNT,
      FIRST,
      LAST,
      BALANCE

   } // end enum

   [STAThread]
   static void Main()
   {
      Application.Run( new BankUIForm() );
   }

   // Visual Studio .NET generated code

   // clear all TextBoxes
   public void ClearTextBoxes()
   {
      // iterate through every Control on form
      for ( int i = 0; i < Controls.Count; i++ )
      {
```

Fig. 17.7 Base class for GUIs in our file-processing applications. (Part 1 of 3.)

```
         Control myControl = Controls[ i ]; // get control

         // determine whether Control is TextBox
         if ( myControl is TextBox )
         {
            // clear Text property (set to empty strng)
            myControl.Text = "";
         }
      }

   } // end method ClearTextBoxes

   // set text box values to string array values
   public void SetTextBoxValues( string[] values )
   {
      // determine whether string array has correct length
      if ( values.Length != TextBoxCount )
      {
         // throw exception if not correct length
         throw( new ArgumentException( "There must be " +
            (TextBoxCount + 1) + " strings in the array" ) );
      }

      // set array values if array has correct length
      else
      {
         // set array values to text box values
         accountTextBox.Text =
            values[ ( int )TextBoxIndices.ACCOUNT ];
         firstNameTextBox.Text =
            values[ ( int )TextBoxIndices.FIRST ];
         lastNameTextBox.Text =
            values[ ( int )TextBoxIndices.LAST ];
         balanceTextBox.Text =
            values[ ( int )TextBoxIndices.BALANCE ];
      }

   } // end method SetTextBoxValues

   // return text box values as string array
   public string[] GetTextBoxValues()
   {
      string[] values = new string[ TextBoxCount ];

      // copy text box fields to string array
      values[ ( int )TextBoxIndices.ACCOUNT ] =
         accountTextBox.Text;
      values[ ( int )TextBoxIndices.FIRST ] =
         firstNameTextBox.Text;
      values[ ( int )TextBoxIndices.LAST ] =
         lastNameTextBox.Text;
      values[ ( int )TextBoxIndices.BALANCE ] =
         balanceTextBox.Text;
```

Fig. 17.7 Base class for GUIs in our file-processing applications. (Part 2 of 3.)

```

         return values;

      } // end method GetTextBoxValues

   } // end class BankUIForm
```



Account

First Name

Last Name

Balance

Fig. 17.7 Base class for GUIs in our file-processing applications. (Part 3 of 3.)

```
// Fig. 17.8: Record.cs
// Serializable class that represents a data record.

using System;

[Serializable]
public class Record
{
   private int account;
   private string firstName;
   private string lastName;
   private double balance;

   // default constructor sets members to default values
   public Record() : this( 0, "", "", 0.0 )
   {
   }

   // overloaded constructor sets members to parameter values
   public Record( int accountValue, string firstNameValue,
      string lastNameValue, double balanceValue )
   {
      Account = accountValue;
      FirstName = firstNameValue;
      LastName = lastNameValue;
      Balance = balanceValue;

   } // end constructor

```

Fig. 17.8 Record for sequential-access file-processing applications. (Part 1 of 3.)

```
      // property Account
      public int Account
      {
         get
         {
            return account;
         }

         set
         {
            account = value;
         }

      } // end property Account

      // property FirstName
      public string FirstName
      {
         get
         {
            return firstName;
         }

         set
         {
            firstName = value;
         }

      } // end property FirstName

      // property LastName
      public string LastName
      {
         get
         {
            return lastName;
         }

         set
         {
            lastName = value;
         }

      } // end property LastName

      // property Balance
      public double Balance
      {
         get
         {
            return balance;
         }

```

Fig. 17.8 Record for sequential-access file-processing applications. (Part 2 of 3.)

```
      set
      {
         balance = value;
      }

   } // end property Balance

} // end class Record
```

Fig. 17.8 Record for sequential-access file-processing applications. (Part 3 of 3.)

Class **`CreateFileForm`** (Fig. 17.9) uses instances of class **`Record`** to create a sequential-access file that might be used in an accounts receivable system—i.e., a program that organizes data regarding money owed by a company's credit clients. For each client, the program obtains an account number and the client's first name, last name and balance (i.e., the amount of money that the client owes to the company for previously received goods or services). The data obtained for each client constitutes a record for that client. In this application, the account number represents the record key—files are created and maintained in account-number order. This program assumes that the user enters records in account-number order. However, a comprehensive accounts receivable system would provide a sorting capability. The user could enter the records in any order, and the records then could be sorted and written to the file in order. (Note that all outputs in this chapter should be read row by row, from left to right in each row.)

Figure 17.9 contains the code for class **`CreateFileForm`**, which either creates or opens a file (depending on whether one exists), then allows the user to write bank information to that file. Line 16 imports the **`BankLibrary`** namespace; this namespace contains class **`BankUIForm`**, from which class **`CreateFileForm`** inherits (line 18). Because of this inheritance relationship, the **`CreateFileForm`** GUI is similar to that of class **`BankUIForm`** (shown in the Fig. 17.9 output), except that the inherited class contains buttons **Save As**, **Enter** and **Exit**.

When the user clicks the **Save As** button, the program invokes method **`saveButton_Click`** (lines 41–85). Line 45 instantiates an object of class **`SaveFileDialog`**, which belongs to the **`System.Windows.Forms`** namespace. Objects of this class are used for selecting files (see the second screen in Fig. 17.9). Line 46 calls method **`ShowDialog`** of the **`SaveFileDialog`** object to display the **`SaveFileDialog`**. When displayed, a **`SaveFileDialog`** prevents the user from interacting with any other window in the program until the user closes the **`SaveFileDialog`** by clicking either **Save** or **Cancel**. Dialogs that behave in this fashion are called *modal dialogs*. The user selects the appropriate drive, directory and file name, then clicks **Save**. Method **`ShowDialog`** returns an integer specifying which button (**Save** or **Cancel**) the user clicked to close the dialog. In this example, the **`Form`** property **`DialogResult`** receives this integer. Line 53 tests whether the user clicked **Cancel** by comparing the value returned by property **`DialogResult`** to constant ***`DialogResult.Cancel`***. If the values are equal, method **`saveButton_Click`** returns (line 54). If the values are unequal (i.e., the user clicked **Save**, instead of clicking **Cancel**), line 57 uses property ***`FileName`*** of class **`SaveFileDialog`** to obtain the user-selected file.

```
// Fig 17.9: CreateSequentialAccessFile.cs
// Creating a sequential-access file.

// C# namespaces
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

// Deitel namespace
using BankLibrary;

public class CreateFileForm : BankUIForm
{
   private System.Windows.Forms.Button saveButton;
   private System.Windows.Forms.Button enterButton;
   private System.Windows.Forms.Button exitButton;

   private System.ComponentModel.Container components = null;

   // serializes Record in binary format
   private BinaryFormatter formatter = new BinaryFormatter();

   // stream through which serializable data is written to file
   private FileStream output;

   [STAThread]
   static void Main()
   {
      Application.Run( new CreateFileForm() );
   }

   // Visual Studio .NET generated code

   // invoked when user clicks Save button
   private void saveButton_Click(
      object sender, System.EventArgs e )
   {
      // create dialog box enabling user to save file
      SaveFileDialog fileChooser = new SaveFileDialog();
      DialogResult result = fileChooser.ShowDialog();
      string fileName; // name of file to save data

      // allow user to create file
      fileChooser.CheckFileExists = false;

```

Fig. 17.9 Create and write to a sequential-access file. (Part 1 of 5.)

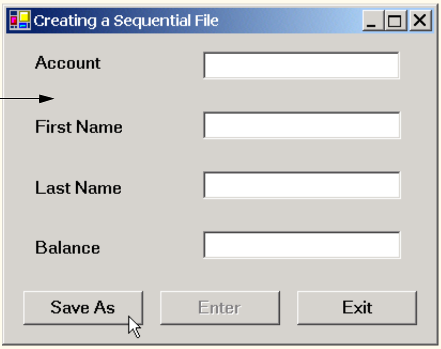
```
      // exit event handler if user clicked "Cancel"
      if ( result == DialogResult.Cancel )
         return;

      // get specified file name
      fileName = fileChooser.FileName;

      // show error if user specified invalid file
      if ( fileName == "" || fileName == null )
         MessageBox.Show( "Invalid File Name", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      else
      {
         // save file via FileStream if user specified valid file
         try
         {
            // open file with write access
            output = new FileStream( fileName,
               FileMode.OpenOrCreate, FileAccess.Write );

            // disable Save button and enable Enter button
            saveButton.Enabled = false;
            enterButton.Enabled = true;
         }

         // handle exception if file does not exist
         catch ( FileNotFoundException )
         {
            // notify user if file does not exist
            MessageBox.Show( "File Does Not Exist", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }
   } // end method saveButton_Click

   // invoke when user clicks Enter button
   private void enterButton_Click(
      object sender, System.EventArgs e )
   {
      // store TextBox values string array
      string[] values = GetTextBoxValues();

      // Record containing TextBox values to serialize
      Record record = new Record();

      // determine whether TextBox account field is empty
      if ( values[ ( int )TextBoxIndices.ACCOUNT ] != "" )
      {
         // store TextBox values in Record and serialize Record
         try
         {
```

Fig. 17.9 Create and write to a sequential-access file. (Part 2 of 5.)

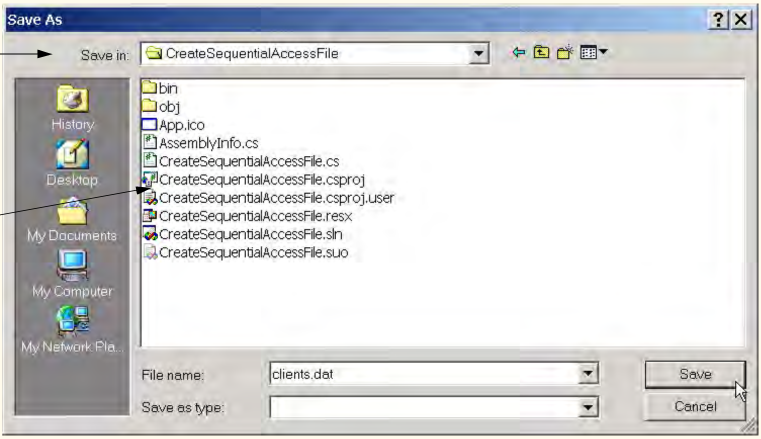
```
             // get account number value from TextBox
             int accountNumber = Int32.Parse(
                values[ ( int )TextBoxIndices.ACCOUNT ] );

             // determine whether accountNumber is valid
             if ( accountNumber > 0 )
             {
                // store TextBox fields in Record
                record.Account = accountNumber;
                record.FirstName =
                   values[ ( int )TextBoxIndices.FIRST ];
                record.LastName =
                   values[ ( int )TextBoxIndices.LAST ];
                record.Balance = Double.Parse( values[
                   ( int )TextBoxIndices.BALANCE ] );

                // write Record to FileStream (serialize object)
                formatter.Serialize( output, record );
             }
             else
             {
                // notify user if invalid account number
                MessageBox.Show( "Invalid Account Number", "Error",
                   MessageBoxButtons.OK, MessageBoxIcon.Error );
             }
          }

          // notify user if error occurs in serialization
          catch( SerializationException )
          {
             MessageBox.Show( "Error Writing to File", "Error",
                MessageBoxButtons.OK, MessageBoxIcon.Error );
          }

          // notify user if error occurs regarding parameter format
          catch( FormatException )
          {
             MessageBox.Show( "Invalid Format", "Error",
                MessageBoxButtons.OK, MessageBoxIcon.Error );
          }
       }

       ClearTextBoxes(); // clear TextBox values

    } // end method enterButton_Click

    // invoked when user clicks Exit button
    private void exitButton_Click(
       object sender, System.EventArgs e )
    {
       // determine whether file exists
       if ( output != null )
       {
```

Fig. 17.9 Create and write to a sequential-access file. (Part 3 of 5.)

```
         // close file
         try
         {
            output.Close();
         }

         // notify user of error closing file
         catch( IOException )
         {
            MessageBox.Show( "Cannot close file", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }

      Application.Exit();

   } // end method exitButton_Click

} // end class CreateFileForm
```


Creating a Sequential File
Account
First Name
Last Name
Balance
Save As
Enter
Exit

BankUI graphical user interface

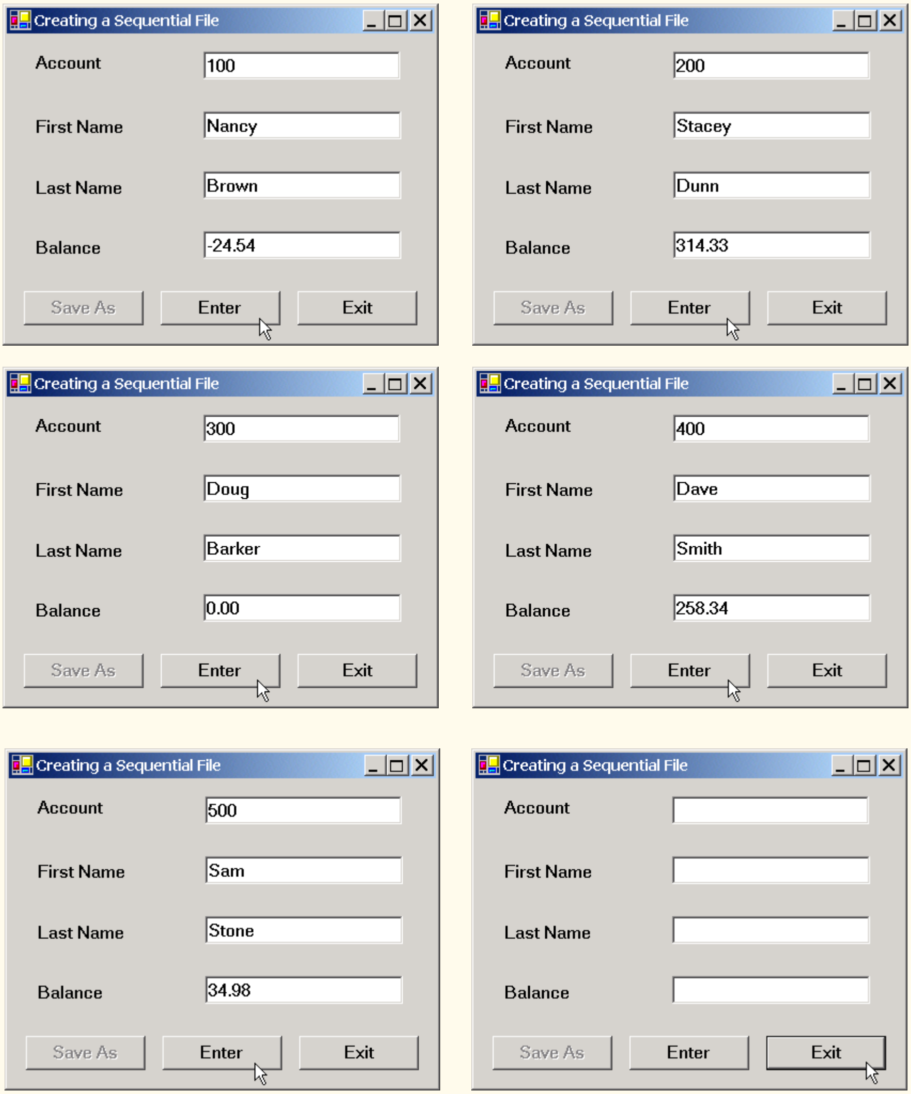

Save As
Save in:
CreateSequentialAccessFile
History
Desktop
My Documents
My Computer
My Network Pla...
bin
obj
App.ico
AssemblyInfo.cs
CreateSequentialAccessFile.cs
CreateSequentialAccessFile.csproj
CreateSequentialAccessFile.csproj.user
CreateSequentialAccessFile.resx
CreateSequentialAccessFile.sln
CreateSequentialAccessFile.suo
Save as type:
Save
Cancel

File name: clients.dat

SaveFileDialog

Files and directories

Fig. 17.9 Create and write to a sequential-access file. (Part 4 of 5.)



Account 100
First Name Nancy
Last Name Brown
Balance -24.54

Account 200
First Name Stacey
Last Name Dunn
Balance 314.33

Account 300
First Name Doug
Last Name Barker
Balance 0.00

Account 400
First Name Dave
Last Name Smith
Balance 258.34

Account 500
First Name Sam
Last Name Stone
Balance 34.98

Account
First Name
Last Name
Balance

Fig. 17.9 Create and write to a sequential-access file. (Part 5 of 5.)

As we stated previously in this chapter, we can open files to perform text manipulation by creating objects of classes **`FileStream`**. In this example, we want the file to be opened for output, so lines 69–70 instantiate a **`FileStream`** object. The **`FileStream`** constructor that we use receives three arguments—a **`string`** containing the name of the file to be opened, a constant describing how to open the file and a constant describing the file permissions. Line 70 passes constant **`FileMode.OpenOrCreate`** to the **`FileStream`** constructor as the constructor's second argument. This constant indicates that the **`FileStream`** object should open the file if the file exists or create the file if the file does not exist. C# offers other **`FileMode`** constants describing how to open files; we introduce these constants as we use them in examples. Line 70 passes constant `FileAccess.Write` to the `FileStream` constructor as the constructor's third argument. This constant ensures that the program can perform write-only operations on the `FileStream` object. C# provides two other constants for this parameter—`FileAccess.Read` for read-only access and `FileAccess.ReadWrite` for both read and write access.

Good Programming Practice 17.1

When opening files, use the `FileAccess` *enumeration to control user access to these files.*

After the user types information in each `TextBox`, the user clicks the **Enter** button, which calls method `enterButton_Click` (lines 88–147) to save data from the `TextBox` in the user-specified file. If the user entered a valid account number (i.e., an integer greater than zero), lines 112–118 store the `TextBox` values in an object of type `Record`. If the user entered invalid data in one of the `TextBox`es (such as entering non-numeric characters in the **Balance** field), the program throws a `FormatException`. The `catch` block in lines 138–142 handles such an exception by notifying the user (via a `MessageBox`) of the improper format. If the user entered valid data, line 120 writes the record to the file by invoking method `Serialize` of the `BinaryFormatter` object (instantiated in line 27). Class `BinaryFormatter` uses methods *`Serialize`* and *`Deserialize`* to write and read objects into streams, respectively. Method `Serialize` writes the object's representation to a file. Method `Deserialize` reads this representation from a file and reconstructs the original object. Both methods throw a `SerializationException` if an error occurs during serialization or deserialization (errors result when the methods attempt to access streams or records that do not exist). Both methods `Serialize` and `Deserialize` require a `Stream` object (e.g., the `FileStream`) as a parameter so that the `BinaryFormatter` can access the correct file; the `BinaryFormatter` must receive an instance of a class that derives from class `Stream`, because `Stream` is `abstract`. Class `BinaryFormatter` belongs to the *`System.Runtime.Serialization.Formatters.Binary`* namespace.

Common Programming Error 17.1

Failure to open a file before attempting to reference it in a program is a logic error.

When the user clicks the **Exit** button, the program invokes method `exitButton_Click` (lines 150–172) to exit the application. Line 159 closes the `FileStream` if one has been opened, and line 170 exits the program.

Performance Tip 17.1

Close each file explicitly when the program no longer needs to reference the file. This can reduce resource usage in programs that continue executing long after they finish using a specific file. The practice of explicitly closing files also improves program clarity.

Performance Tip 17.2

Releasing resources explicitly when they are no longer needed makes them immediately available for reuse by the program, thus improving resource utilization.

In the sample execution for the program in Fig. 17.9, we entered information for five accounts (Fig. 17.10). The program does not depict how the data records are rendered in

Account Number	**First Name**	**Last Name**	**Balance**
`100`	`Nancy`	`Brown`	`-25.54`
`200`	`Stacey`	`Dunn`	`314.33`
`300`	`Doug`	`Barker`	`0.00`
`400`	`Dave`	`Smith`	`258.34`
`500`	`Sam`	`Stone`	`34.98`

Fig. 17.10 Sample data for the program of Fig. 17.9.

the file. To verify that the file has been created successfully, in the next section, we create a program to read and display the file.

17.6 Reading Data from a Sequential-Access File

Data are stored in files so that they can be retrieved for processing when they are needed. The previous section demonstrated how to create a file for use in sequential-access applications. In this section, we discuss how to read (or retrieve) data sequentially from a file.

Class `ReadSequentialAccessFileForm` (Fig. 17.11) reads records from the file created by the program in Fig. 17.9, then displays the contents of each record. Much of the code in this example is similar to that of Fig. 17.9, so we discuss only the unique aspects of the application.

When the user clicks the **Open File** button, the program calls method `openButton_Click` (lines 40–70). Line 44 instantiates an object of class *`OpenFileDialog`*, and line 45 calls the object's *`ShowDialog`* method to display the **Open** dialog (see the second screenshot in Fig. 17.11). The behavior and GUI for the two dialog types are the same (except that **Save** is replaced by **Open**). If the user inputs a valid file name, lines 63–64 create a `FileStream` object and assign it to reference `input`. We pass constant `FileMode.Open` as the second argument to the `FileStream` constructor. This constant indicates that the `FileStream` should open the file if the file exists or should throw a `FileNotFoundException` if the file does not exist. (In this example, the `FileStream` constructor will not throw a `FileNotFoundException`, because the `OpenFileDialog` requires the user to enter a name of a file that exists.) In the last example (Fig. 17.9), we wrote text to the file using a `FileStream` object with write-only access. In this example, (Fig. 17.11), we specify read-only access to the file by passing constant `FileAccess.Read` as the third argument to the `FileStream` constructor.

Testing and Debugging Tip 17.1

Open a file with the `FileAccess.Read` file-open mode if the contents of the file should not be modified. This prevents unintentional modification of the file's contents.

When the user clicks the **Next Record** button, the program calls method `nextButton_Click` (lines 73–113), which reads the next record from the user-specified file. (The user must click **Next Record** after opening the file to view the first record.) Lines 80–81 call method `Deserialize` of the `BinaryFormatter` object to read the next record. Method `Deserialize` reads the data and casts the result to a `Record`—this cast is necessary, because **`Deserialize`** returns a reference of type **`Object`**. Lines 84–91 then display the **`Record`** values in the **`TextBox`**es. When method **`Deserialize`** attempts to deserialize a record that does not exist in the file (i.e., the program has displayed all file records), the method throws a **`SerializationException`**. The **`catch`** block (lines 95–111) that handles this exception closes the **`FileStream`** object (line 98) and notifies the user that there are no more records (lines 109–110).

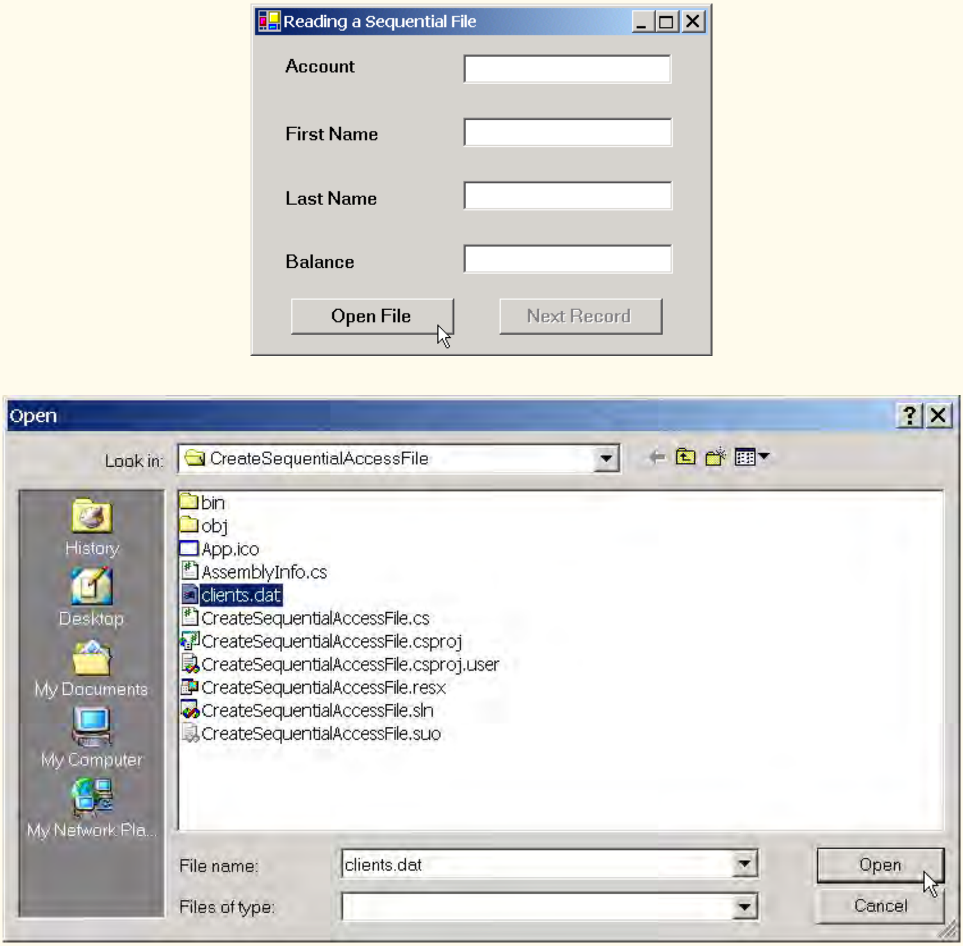
```
 1  // Fig. 17.11: ReadSequentialAccessFile.cs
 2  // Reading a sequential-access file.
 3
 4  // C# namespaces
 5  using System;
 6  using System.Drawing;
 7  using System.Collections;
 8  using System.ComponentModel;
 9  using System.Windows.Forms;
10  using System.Data;
11  using System.IO;
12  using System.Runtime.Serialization.Formatters.Binary;
13  using System.Runtime.Serialization;
14
15  // Deitel namespaces
16  using BankLibrary;
17
18  public class ReadSequentialAccessFileForm : BankUIForm
19  {
20     System.Windows.Forms.Button openButton;
21     System.Windows.Forms.Button nextButton;
22
23     private System.ComponentModel.Container components = null;
24
25     // stream through which serializable data are read from file
26     private FileStream input;
27
28     // object for deserializing Record in binary format
29     private BinaryFormatter reader = new BinaryFormatter();
30
31     [STAThread]
32     static void Main()
33     {
34        Application.Run( new ReadSequentialAccessFileForm() );
35     }
36
37     // Visual Studio .NET generated code
38
39     // invoked when user clicks Open button
40     private void openButton_Click(
41        object sender, System.EventArgs e )
42     {
43        // create dialog box enabling user to open file
44        OpenFileDialog fileChooser = new OpenFileDialog();
45        DialogResult result = fileChooser.ShowDialog();
```

Fig. 17.11 Reading sequential-access files. (Part 1 of 4.)

```
      string fileName; // name of file containing data

      // exit event handler if user clicked Cancel
      if ( result == DialogResult.Cancel )
         return;

      // get specified file name
      fileName = fileChooser.FileName;
      ClearTextBoxes();

      // show error if user specified invalid file
      if ( fileName == "" || fileName == null )
         MessageBox.Show( "Invalid File Name", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      else
      {
         // create FileStream to obtain read access to file
         input = new FileStream( fileName, FileMode.Open,
            FileAccess.Read );

         // enable next record button
         nextButton.Enabled = true;
      }

   } // end method openButton_Click

   // invoked when user clicks Next button
   private void nextButton_Click(
      object sender, System.EventArgs e )
   {
      // deserialize Record and store data in TextBoxes
      try
      {
         // get next Record available in file
         Record record =
            ( Record )reader.Deserialize( input );

         // store Record values in temporary string array
         string[] values = new string[] {
            record.Account.ToString(),
            record.FirstName.ToString(),
            record.LastName.ToString(),
            record.Balance.ToString() };

         // copy string array values to TextBox values
         SetTextBoxValues( values );
      }

      // handle exception when no Records in file
      catch( SerializationException )
      {\
         // close FileStream if no Records in file
         input.Close();
```

Fig. 17.11 Reading sequential-access files. (Part 2 of 4.)

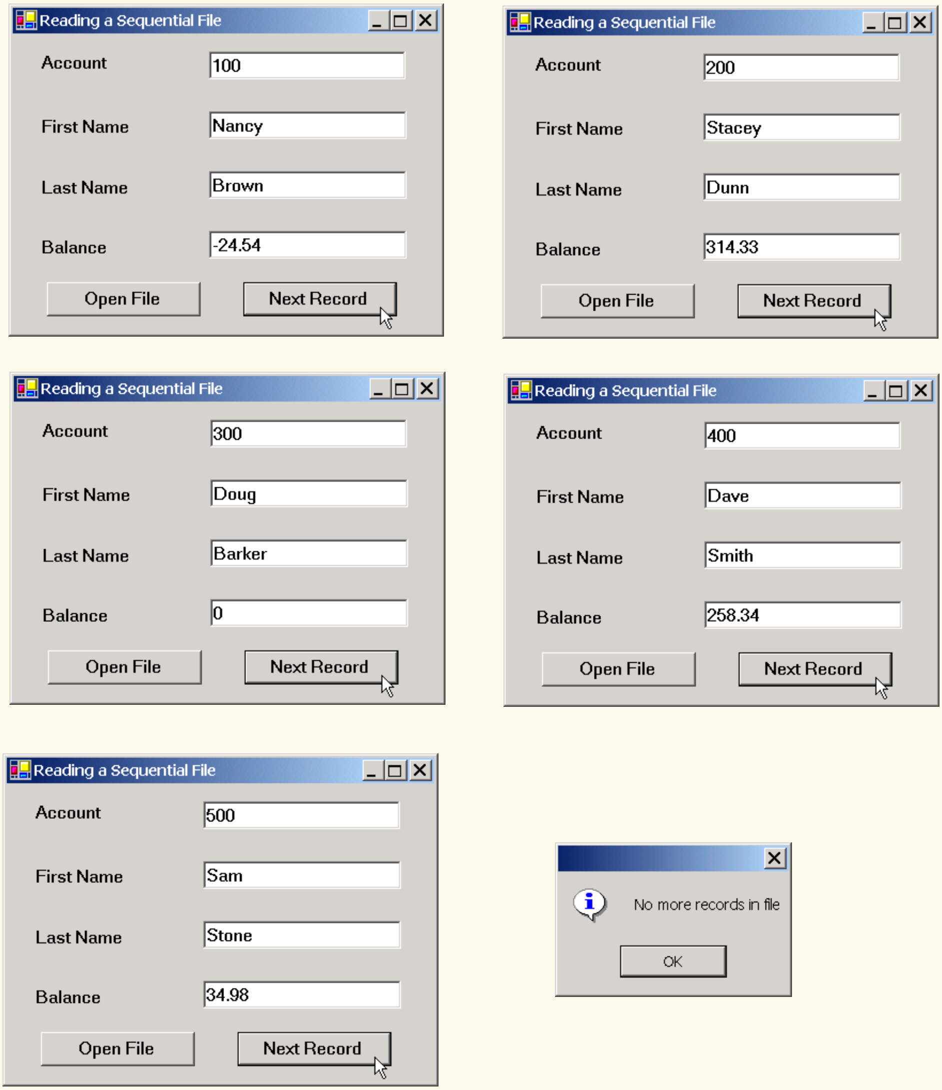
```

         // enable Open Record button
         openButton.Enabled = true;

         // disable Next Record button
         nextButton.Enabled = false;

         ClearTextBoxes();

         // notify user if no Records in file
         MessageBox.Show( "No more records in file", "",
            MessageBoxButtons.OK, MessageBoxIcon.Information );
      }

   } // end method nextButton_Click

} // end class ReadSequentialAccessFileForm
```


Reading a Sequential File
Account
First Name
Last Name
Balance
Open File
Next Record
Open
Look in:
History
Desktop
My Documents
My Computer
My Network Pla...
bin
obj
App.ico
AssemblyInfo.cs
clients.dat
CreateSequentialAccessFile.cs
CreateSequentialAccessFile.csproj
CreateSequentialAccessFile.csproj.user
CreateSequentialAccessFile.resx
CreateSequentialAccessFile.sln
CreateSequentialAccessFile.suo
Files of type:
Open
Cancel

Look in: CreateSequentialAccessFile

File name: clients.dat

Fig. 17.11 Reading sequential-access files. (Part 3 of 4.)



Account 100
First Name Nancy
Last Name Brown
Balance -24.54

Account 200
First Name Stacey
Last Name Dunn
Balance 314.33

Account 300
First Name Doug
Last Name Barker
Balance 0

Account 400
First Name Dave
Last Name Smith
Balance 258.34

Account 500
First Name Sam
Last Name Stone
Balance 34.98

No more records in file

Fig. 17.11 Reading sequential-access files. (Part 4 of 4.)

To retrieve data sequentially from a file, programs normally start from the beginning of the file, reading data consecutively until the desired data are found. It sometimes is necessary to process a file sequentially several times (from the beginning of the file) during the execution of a program. A **`FileStream`** object can reposition its *file-position pointer* (which contains the byte number of the next byte to be read from or written to the file) to any position in the file—we show this feature when we introduce random-access file-processing applications. When a **`FileStream`** object is opened, its file-position pointer is set to zero (i.e., the beginning of the file)

Performance Tip 17.3

It is time-consuming to close and reopen a file for the purpose of moving the file-position pointer to the file's beginning. Doing so frequently could slow program performance.

We now present a more substantial program that builds on the concepts employed in Fig. 17.11. Class `creditInquiryForm` (Fig. 17.12) is a credit-inquiry program that enables a credit manager to display account information for those customers with credit balances (i.e., customers to whom the company owes money), zero balances (i.e., customers who do not owe the company money) and debit balances (i.e., customers who owe the company money for previously received goods and services). Note that line 21 declares a *`RichTextBox`* that will display the account information. `RichTextBox`es provide more functionality than do regular `TextBox`es—for example, `RichTextBox`es offer method *`Find`* for searching individual strings and method *`LoadFile`* for displaying file contents. Class `RichTextBox` does not inherit from class `TextBox`; rather, both classes inherit directly from `abstract` class *`System.Windows.Forms.TextBoxBase`*. We use a `RichTextBox` in this example, because a `RichTextBox` displays multiple lines of text by default, whereas a regular `TextBox` displays only one. Alternatively, we could have specified that a `TextBox` object display multiple lines of text by setting its `Multiline` property to `true`

The program in Fig. 17.12 displays buttons that enable a credit manager to obtain credit information. The **Open File** button opens a file for gathering data. The **Credit Balances** button displays a list of accounts that have credit balances, the **Debit Balances** button displays a list of accounts that have debit balances, and the **Zero Balances** button displays a list of accounts that have zero balances. The **Done** button exits the application.

When the user clicks the **Open File** button, the program calls method `openButton_Click` (lines 49–76). Line 53 instantiates an object of class *`OpenFileDialog`*, and line 54 calls the object's *`ShowDialog`* method to display the **Open** dialog, in which the user inputs the name of the file to open. .

When the user clicks **Credit Balances**, **Debit Balances** or **Zero Balances**, the program invokes method `get_Click` (lines 80–142). Line 83 casts the `sender` parameter, which is a reference to the object that sent the event, to a `Button` object. Line 86 extracts the `Button` object's text, which the program uses to determine which GUI `Button` the user clicked. Lines 96–97 create a `FileStream` object with read-only file access and assign it to reference `input`. Lines 102–125 define a `while` loop that uses `private` method `ShouldDisplay` (lines 145–170) to determine whether to display each record in the file. The `while` loop obtains each record by calling method `Deserialize` of the `FileStream` object repeatedly (line 105). When the file-position pointer reaches the end of file, method `Deserialize` throws a `SerializationException`, which the `catch` block in lines 136–140 handles: Line 139 calls the `Close` method of `FileStream` to close the file, and method `get_Click` returns.

```
1  // Fig. 17.12: CreditInquiry.cs
2  // Read a file sequentially and display contents based on
3  // account type specified by user (credit, debit or zero balances).
4
```

Fig. 17.12 Credit-inquiry program. (Part 1 of 7.)

```
// C# namespaces
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

// Deitel namespaces
using BankLibrary;

public class CreditInquiryForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.RichTextBox displayTextBox;

   private System.Windows.Forms.Button doneButton;
   private System.Windows.Forms.Button zeroButton;
   private System.Windows.Forms.Button debitButton;
   private System.Windows.Forms.Button creditButton;
   private System.Windows.Forms.Button openButton;

   private System.ComponentModel.Container components = null;

   // stream through which serializable data are read from file
   private FileStream input;

   // object for deserializing Record in binary format
   BinaryFormatter reader = new BinaryFormatter();

   // name of file that stores credit, debit and zero balances
   private string fileName;

   [STAThread]
   static void Main()
   {
      Application.Run( new CreditInquiryForm() );
   }

   // Visual Studio .NET generated code

   // invoked when user clicks Open File button
   private void openButton_Click(
      object sender, System.EventArgs e )
   {
      // create dialog box enabling user to open file
      OpenFileDialog fileChooser = new OpenFileDialog();
      DialogResult result = fileChooser.ShowDialog();

```

Fig. 17.12 Credit-inquiry program. (Part 2 of 7.)

```
         // exit event handler if user clicked Cancel
         if ( result == DialogResult.Cancel )
            return;

         // get name from user
         fileName = fileChooser.FileName;

         // show error if user specified invalid file
         if ( fileName == "" || fileName == null )
            MessageBox.Show( "Invalid File Name", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         else
         {
            // enable all GUI buttons, except for Open file button
            openButton.Enabled = false;
            creditButton.Enabled = true;
            debitButton.Enabled = true;
            zeroButton.Enabled = true;
         }

      } // end method openButton_Click

      // invoked when user clicks credit balances,
      // debit balances or zero balances button
      private void get_Click( object sender, System.EventArgs e )
      {
         // convert sender explicitly to object of type button
         Button senderButton = ( Button )sender;

         // get text from clicked Button, which stores account type
         string accountType = senderButton.Text;

         // read and display file information
         try
         {
            // close file from previous operation
            if ( input != null )
               input.Close();

            // create FileStream to obtain read access to file
            input = new FileStream( fileName, FileMode.Open,
               FileAccess.Read );

            displayTextBox.Text = "The accounts are:\r\n";

            // traverse file until end of file
            while ( true )
            {
               // get next Record available in file
               Record record = ( Record )reader.Deserialize( input );

               // store record's last field in balance
               Double balance = record.Balance;
```

Fig. 17.12 Credit-inquiry program. (Part 3 of 7.)

```

                  // determine whether to display balance
                  if ( ShouldDisplay( balance, accountType ) )
                  {
                     // display record
                     string output = record.Account + "\t" +
                        record.FirstName + "\t" + record.LastName +
                        new string( ' ', 6 ) + "\t";

                     // display balance with correct monetary format
                     output += String.Format(
                        "{0:F}", balance ) + "\r\n";

                     // copy output to screen
                     displayTextBox.Text += output;
                  }
               }
            }

            // handle exception when file cannot be closed
            catch( IOException )
            {
               MessageBox.Show( "Cannot Close File", "Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error );
            }

            // handle exception when no more records
            catch( SerializationException )
            {
               // close FileStream if no Records in file
               input.Close();
            }

         } // end method get_Click

         // determine whether to display given record
        private bool ShouldDisplay( double balance, string accountType )
         {
            if ( balance > 0 )
            {
               // display credit balances
               if ( accountType == "Credit Balances" )
                  return true;
            }

            else if ( balance < 0 )
            {
               // display debit balances
               if ( accountType == "Debit Balances" )
                  return true;
            }

```

Fig. 17.12 Credit-inquiry program. (Part 4 of 7.)

```
      else // balance == 0
      {
         // display zero balances
         if ( accountType == "Zero Balances" )
            return true;
      }

      return false;

   } // end method ShouldDisplay

   // invoked when user clicks Done button
   private void doneButton_Click(
      object sender, System.EventArgs e )
   {
      // determine whether file exists
      if ( input != null )
      {
         // close file
         try
         {
            input.Close();
         }

         // handle exception if FileStream does not exist
         catch( IOException )
         {
            // notify user of error closing file
            MessageBox.Show( "Cannot close file", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error);
         }
      }

      Application.Exit();

   } // end method doneButton_Click

} // end class CreditInquiryForm
```

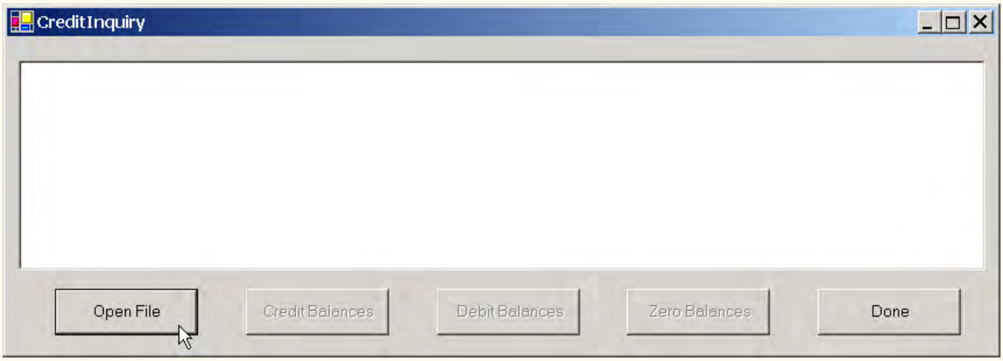
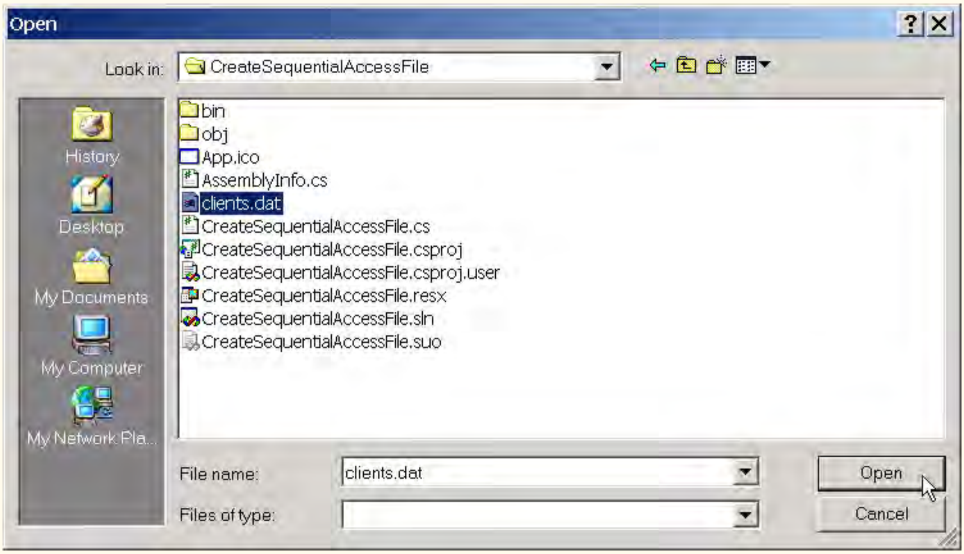
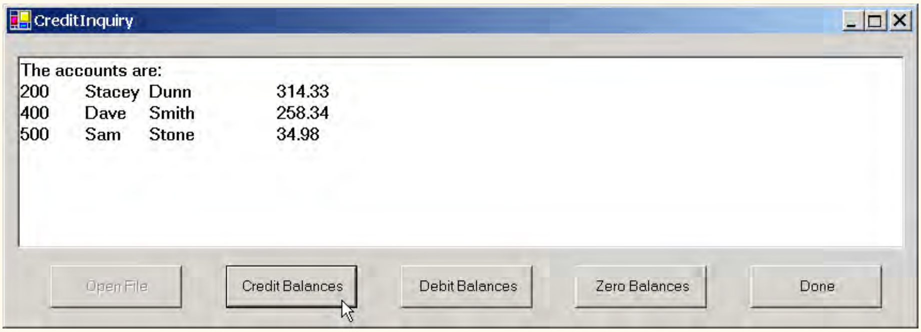

CreditInquiry
Open File
Credit Balances
Debit Balances
Zero Balances
Done

Fig. 17.12 Credit-inquiry program. (Part 5 of 7.)

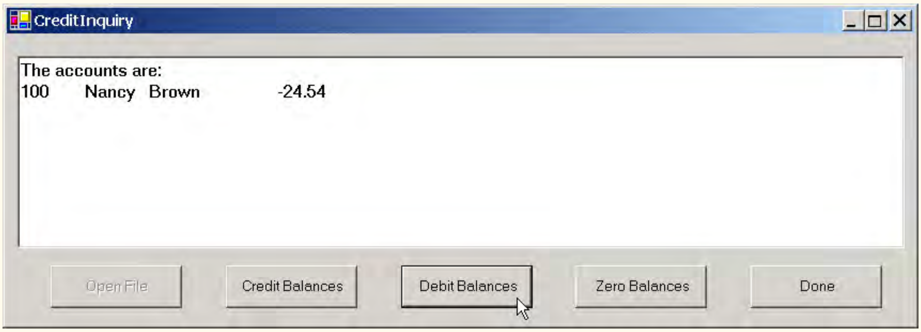

Open
Look in:
CreateSequentialAccessFile
History
Desktop
My Documents
My Computer
My Network Pla...
Files of type:
Open
Cancel

bin
obj
App.ico
AssemblyInfo.cs
clients.dat
CreateSequentialAccessFile.cs
CreateSequentialAccessFile.csproj
CreateSequentialAccessFile.csproj.user
CreateSequentialAccessFile.resx
CreateSequentialAccessFile.sln
CreateSequentialAccessFile.suo

File name: clients.dat

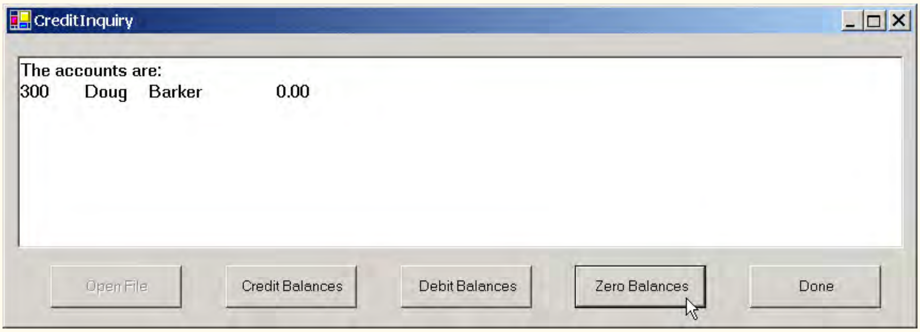

CreditInquiry
Open File
Credit Balances
Debit Balances
Zero Balances
Done

```
The accounts are:
200     Stacey  Dunn          314.33
400     Dave    Smith         258.34
500     Sam     Stone         34.98
```


CreditInquiry
Open File
Credit Balances
Debit Balances
Zero Balances
Done

```
The accounts are:
100     Nancy   Brown         -24.54
```

Fig. 17.12 Credit-inquiry program. (Part 6 of 7.)



```
The accounts are:
300     Doug    Barker          0.00
```

Fig. 17.12 Credit-inquiry program. (Part 7 of 7.)

17.7 Random-Access Files

So far, we have explained how to create sequential-access files and how to search through such files to locate particular information. However, sequential-access files are inappropriate for so-called *"instant-access" applications*, in which a particular record of information must be located immediately. Popular instant-access applications include airline-reservation systems, banking systems, point-of-sale systems, automated-teller machines and other kinds of *transaction-processing systems* requiring rapid access to specific data. The bank at which an individual has an account might have hundreds of thousands or even millions of other customers; however, when that individual uses an automated teller machine, the appropriate account is checked for sufficient funds in seconds. This type of instant access is made possible by *random-access files*. Individual records of a random-access file can be accessed directly (and quickly), without searching through potentially large numbers of other records, as is necessary with sequential-access files. Random-access files sometimes are called *direct-access files*.

As we discussed earlier in this chapter, C# does not impose structure on files, so applications that use random-access files must implement the random-access capability. There are a variety of techniques for creating random-access files. Perhaps the simplest involves requiring that all records in a file be of a uniform, fixed length. The use of fixed-length records enables a program to calculate (as a function of the record size and the record key) the exact location of any record in relation to the beginning of the file. We soon demonstrate how this facilitates immediate access to specific records, even in large files.

Figure 17.13 illustrates the organization of a random-access file composed of fixed-length records (each record in this figure is 100 bytes long). Students can consider a random-access file as analogous to a railroad train with many cars, some of which are empty and some of which contain contents.

Data can be inserted into a random-access file without destroying other data in the file. In addition, previously stored data can be updated or deleted without rewriting the entire file. In the following sections, we explain how to create a random-access file, write data to that file, read data both sequentially and randomly, update data and delete data that is no longer needed.

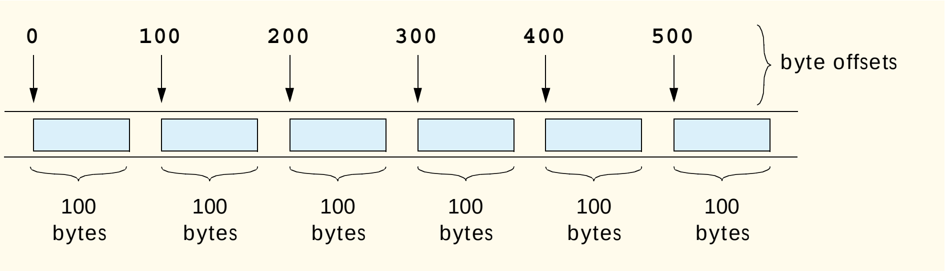


Fig. 17.13 Random-access file with fixed-length records.

Figure 17.14 contains class **`RandomAccessRecord`**, which is used in the random-access file-processing applications in this chapter. This class also belongs to the **`BankLibrary`** DLL—i.e., it is part of the project that contains classes **`BankUIForm`** and **`Record`**. (When adding class **`RandomAccessRecord`** to the project containing **`BankUIForm`** and **`Record`**, remember to rebuild the project.)

Like class **`Record`** (Fig. 17.8), class **`RandomAccessRecord`** contains **`private`** data members (lines 20–23) for storing record information, two constructors for setting these members to default and parameter-specified values, respectively, and properties for accessing these members. However, class **`RandomAccessRecord`** does not contain attribute **`[Serializable]`** before its class definition. We do not serialize this class, because C# does not provide a means to obtain an object's size at runtime. This means that, if we serialize the class, we cannot guarantee a fixed-length record size.

Instead of serializing the class, we fix the length of the **`private`** data members, then write those data as a byte stream to the file. To fix this length, the **`set`** accessors of properties **`FirstName`** (lines 58–91) and **`LastName`** (lines 94–127) ensure that members **`firstName`** and **`lastName`** are **`char`** arrays of exactly 15 elements. Each **`set`** accessor receives as an argument a **`string`** representing the first name and last name, respectively. If the **`string`** parameter contains fewer than 15 characters, the property's **`set`** accessor copies the **`string`**'s values to the **`char`** array, then populates the remainder with spaces. If the **`string`** parameter contains more than 15 characters, the **`set`** accessor stores only the first 15 characters of the **`string`** parameter into the **`char`** array.

```
 1  // Fig. 17.14: RandomAccessRecord.cs
 2  // Data-record class for random-access applications.
 3  
 4  using System;
 5  
 6  public class RandomAccessRecord
 7  {
 8     // length of firstName and lastName
 9     private const int CHAR_ARRAY_LENGTH = 15;
10  
11     private const int SIZE_OF_CHAR = 2;
12     private const int SIZE_OF_INT32 = 4;
13     private const int SIZE_OF_DOUBLE = 8;
```

Fig. 17.14 Record for random-access file-processing applications. (Part 1 of 4.)

```

   // length of record
   public const int SIZE = SIZE_OF_INT32 +
      2 * ( SIZE_OF_CHAR * CHAR_ARRAY_LENGTH ) + SIZE_OF_DOUBLE;

   // record data
   private int account;
   private char[] firstName = new char[ CHAR_ARRAY_LENGTH ];
   private char[] lastName = new char[ CHAR_ARRAY_LENGTH ];
   private double balance;

   // default constructor sets members to default values
   public RandomAccessRecord() : this( 0, "", "", 0.0 )
   {
   }

   // overloaded counstructor sets members to parameter values
   public RandomAccessRecord( int accountValue,
      string firstNameValue, string lastNameValue,
      double balanceValue )
   {
      Account = accountValue;
      FirstName = firstNameValue;
      LastName = lastNameValue;
      Balance = balanceValue;

   } // end constructor

   // property Account
   public int Account
   {
      get
      {
         return account;
      }

      set
      {
         account = value;
      }

   } // end property Account

   // property FirstName
   public string FirstName
   {
      get
      {
         return new string( firstName );
      }

```

Fig. 17.14 Record for random-access file-processing applications. (Part 2 of 4.)

```
         set
         {
            // determine length of string parameter
            int stringSize = value.Length;

            // firstName string representation
            string firstNameString = value;

            // append spaces to string parameter if too short
            if ( CHAR_ARRAY_LENGTH >= stringSize )
            {
               firstNameString = value +
                  new string( ' ', CHAR_ARRAY_LENGTH - stringSize );
            }
            else
            {
               // remove characters from string parameter if too long
               firstNameString =
                  value.Substring( 0, CHAR_ARRAY_LENGTH );
            }

            // convert string parameter to char array
            firstName = firstNameString.ToCharArray();

         } // end set

      } // end property FirstName

      // property LastName
      public string LastName
      {
         get
         {
            return new string( lastName );
         }

         set
         {
            // determine length of string parameter
            int stringSize = value.Length;

            // lastName string representation
            string lastNameString = value;

            // append spaces to string parameter if too short
            if ( CHAR_ARRAY_LENGTH >= stringSize )
            {
               lastNameString = value +
                  new string( ' ', CHAR_ARRAY_LENGTH - stringSize );
            }
```

Fig. 17.14 Record for random-access file-processing applications. (Part 3 of 4.)

```
115          else
116          {
117             // remove characters from string parameter if too long
118             lastNameString =
119                value.Substring( 0, CHAR_ARRAY_LENGTH );
120          }
121
122          // convert string parameter to char array
123          lastName = lastNameString.ToCharArray();
124
125       } // end set
126
127    } // end property LastName
128
129    // property Balance
130    public double Balance
131    {
132       get
133       {
134          return balance;
135       }
136
137       set
138       {
139          balance = value;
140       }
141
142    } // end property Balance
143
144 } // end class RandomAccessRecord
```

Fig. 17.14 Record for random-access file-processing applications. (Part 4 of 4.)

Lines 16–17 declare **`const SIZE`**, which specifies the record's length. Each record contains **`account`** (4-byte **`int`**), **`firstName`** and **`lastName`** (two 15-element **`char`** arrays, where each **`char`** occupies two bytes, resulting in a total of 60 bytes) and **`balance`** (8-byte **`double`**). In this example, each record (i.e., the four **`private`** data members that our programs will read to and write from files) occupies 72 bytes (4 bytes + 60 bytes + 8 bytes).

17.8 Creating a Random-Access File

Consider the following problem statement for a credit-processing application:

> *Create a transaction-processing program capable of storing a maximum of 100 fixed-length records for a company that can have a maximum of 100 customers. Each record consists of an account number (which acts as the record key), a last name, a first name and a balance. The program can update an account, create an account and delete an account.*

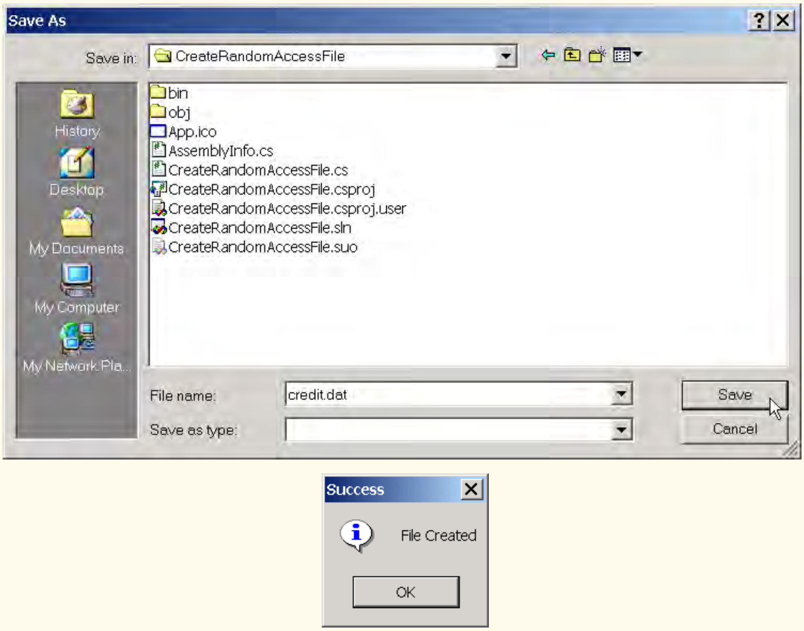
The next several sections introduce the techniques necessary to create this credit-processing program. We now discuss the program used to create the random-access file that the programs of Fig. 17.16 and Fig. 17.17 and the transaction-processing application use to manipulate data. Class **`CreateRandomAccessFile`** (Fig. 17.15) creates a random-access file.

```
// Fig. 17.15: CreateRandomAccessFile.cs
// Creating a random file.

// C# namespaces
using System;
using System.IO;
using System.Windows.Forms;

// Deitel namespaces
using BankLibrary;

class CreateRandomAccessFile
{
   // number of records to write to disk
   private const int NUMBER_OF_RECORDS = 100;

   [STAThread]
   static void Main(string[] args)
   {
      // create random file, then save to disk
      CreateRandomAccessFile file = new CreateRandomAccessFile();
      file.SaveFile();

   } // end method Main

   // write records to disk
   private void SaveFile()
   {
      // record for writing to disk
      RandomAccessRecord blankRecord = new RandomAccessRecord();

     // stream through which serializable data are written to file
      FileStream fileOutput = null;

      // stream for writing bytes to file
      BinaryWriter binaryOutput = null;

      // create dialog box enabling user to save file
      SaveFileDialog fileChooser = new SaveFileDialog();
      DialogResult result = fileChooser.ShowDialog();

      // get file name from user
      string fileName = fileChooser.FileName;

      // exit event handler if user clicked Cancel
      if ( result == DialogResult.Cancel )
         return;

      // show error if user specified invalid file
      if ( fileName == "" || fileName == null )
         MessageBox.Show("Invalid File Name", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
```

Fig. 17.15 Creating files for random-access file-processing applications. (Part 1 of 3.)

```
      else
      {
         // write records to file
         try
         {
            // create FileStream to hold records
            fileOutput = new FileStream( fileName,
               FileMode.Create, FileAccess.Write );

            // set length of file
            fileOutput.SetLength( RandomAccessRecord.SIZE *
               NUMBER_OF_RECORDS );

            // create object for writing bytes to file
            binaryOutput = new BinaryWriter( fileOutput );

            // write empty records to file
            for ( int i = 0; i < NUMBER_OF_RECORDS; i++ )
            {
               // set file position pointer in file
               fileOutput.Position = i * RandomAccessRecord.SIZE;

               // write blank record to file
               binaryOutput.Write( blankRecord.Account );
               binaryOutput.Write( blankRecord.FirstName );
               binaryOutput.Write( blankRecord.LastName );
               binaryOutput.Write( blankRecord.Balance );
            }

            // notify user of success
            MessageBox.Show("File Created", "Success",
               MessageBoxButtons.OK, MessageBoxIcon.Information);
         }

         // handle exception if error occurs during writing
         catch( IOException )
         {
            // notify user of error
            MessageBox.Show( "Cannot write to file", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }

      // close FileStream
      if ( fileOutput == null )
         fileOutput.Close();

      // close BinaryWriter
      if ( binaryOutput == null )
         binaryOutput.Close();

   } // end method SaveFile
} // end class CreateRandomAccessFile
```

Fig. 17.15 Creating files for random-access file-processing applications. (Part 2 of 3.)


Save As
Save in: CreateRandomAccessFile
bin
obj
App.ico
AssemblyInfo.cs
CreateRandomAccessFile.cs
CreateRandomAccessFile.csproj
CreateRandomAccessFile.csproj.user
CreateRandomAccessFile.sln
CreateRandomAccessFile.suo
History
Desktop
My Documents
My Computer
My Network Pla...
Save as type:
Save
Cancel
Success
OK

File name: credit.dat

File Created

Fig. 17.15 Creating files for random-access file-processing applications. (Part 3 of 3.)

Method **`Main`** (lines 18–24) starts the application, which creates a random-access file by calling user-defined method **`SaveFile`** (lines 27–104). Method **`SaveFile`** populates a file with 100 copies of the default (i.e., empty) values for **`private`** data members **`account`**, **`firstName`**, **`lastName`** and **`balance`** of class **`RandomAccessRecord`**. Lines 39–40 create and display the **`SaveFileDialog`**, which enables a user to specify the file to which the program writes data. Using this file, lines 59–60 instantiate the **`FileStream`**. Note that line 60 passes constant **`FileMode.Create`**, which either creates the specified file, if the file does not exist, or overwrites the specified file if it does exist. Lines 63–64 sets the **`FileStream`**'s length, which is equal to the size of an individual **`RandomAccessRecord`** (obtained through constant **`RandomAccessRecord.SIZE`**) multiplied by the number of records we want to copy (obtained through constant **`NUMBER_OF_RECORDS`** in line 15, which we set to value **`100`**).

We now require a means to write bytes to a file. Class ***`BinaryWriter`*** of namespace **`System.IO`** provides methods for writing bytes to streams. The **`BinaryWriter`** constructor takes as an argument a reference to an instance of class **`System.IO.Stream`**, through which the **`BinaryWriter`** can write bytes. Class **`FileStream`** provides methods for writing streams to files and inherits from class **`Stream`**, so we can pass the **`FileStream`** object as an argument to the **`BinaryWriter`** constructor (line 67). Now, we can use the **`BinaryWriter`** to write bytes directly to the file.

Lines 70–80 populate the file with 100 copies of the empty record values (i.e., default values for **`private`** data members of class **`RandomAccessRecord`**). Line 73 changes the file-position pointer to specify the location in the file at which to write the next empty record. Now that we are working with a random-access file, we must set the file-pointer explicitly, using the **`FileStream`** object's **`Position`** property. This property receives as an argument a `long` value describing where to position the pointer relative to the beginning of the file—in this example, we set the pointer so that it advances a number of bytes that is equal to the record size (obtained by `RandomAccessRecord.SIZE`). Lines 76–79 call method `Write` of the `BinaryWriter` object to write the data. Method `Write` is an overloaded method that receives as an argument any primitive data type, then writes that type to a stream of bytes. After the `for` loop exits, lines 97–102 close the `FileStream` and `BinaryWriter` objects.

17.9 Writing Data Randomly to a Random-Access File

Now that we have created a random-access file, we use class `WriteRandomAccessFileForm` (Fig. 17.16) to write data to that file. When a user clicks the **Open File** button, the program invokes method `openButton_Click` (lines 41–84), which displays the `OpenFileDialog` for specifying the file in which to serialize data (lines 45–46); the program then uses the specified file to create a `FileStream` object with write-only access (lines 65–66). Line 69 uses the `FileStream` reference to instantiate an object of class `BinaryWriter`, enabling the program to write bytes to files. We used the same approach when working with class `CreateRandomAccessFile` (Fig. 17.15).

The user enters values in the `TextBox`es for the account number, first name, last name and balance. When the user clicks the **Enter** button, the program invokes method `enterButton_Click` (lines 87–139), which writes the data in the `TextBox`es to the file. Line 91 calls method `GetTextBoxValues` (provided by base class `BankUIForm`) to retrieve the data. Lines 104–105 determine whether the **Account Number** `TextBox` holds valid information (i.e., the account number is in the `1`–`100` range).

```
 1  // Fig 17.16: WriteRandomAccessFile.cs
 2  // Write data to a random-access file.
 3
 4  // C# namespaces
 5  using System;
 6  using System.Drawing;
 7  using System.Collections;
 8  using System.ComponentModel;
 9  using System.Windows.Forms;
10  using System.Data;
11  using System.IO;
12
13  // Deitel namespaces
14  using BankLibrary;
15
16  public class WriteRandomAccessFileForm : BankUIForm
17  {
18     private System.Windows.Forms.Button openButton;
19     private System.Windows.Forms.Button enterButton;
20
21     private System.ComponentModel.Container components = null;
22
```

Fig. 17.16 Writing records to random-access files. (Part 1 of 5.)

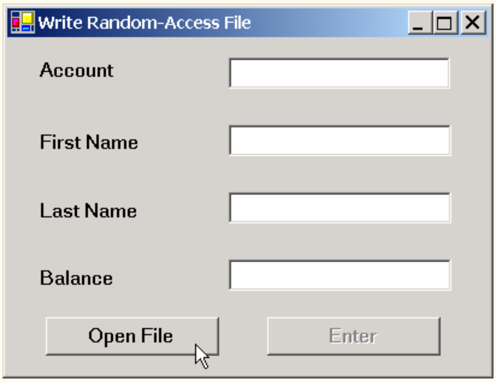
```
      // number of RandomAccessRecords to write to disk
      private const int NUMBER_OF_RECORDS = 100;

      // stream through which data are written to file
      private FileStream fileOutput;

      // stream for writing bytes to file
      private BinaryWriter binaryOutput;

      [STAThread]
      static void Main()
      {
         Application.Run( new WriteRandomAccessFileForm() );
      }

      // Visual Studio .NET generated code

      // invoked when user clicks Open button
      private void openButton_Click(
         object sender, System.EventArgs e )
      {
         // create dialog box enabling user to open file
         OpenFileDialog fileChooser = new OpenFileDialog();
         DialogResult result = fileChooser.ShowDialog();

         // get file name from user
         string fileName = fileChooser.FileName;

         // exit event handler if user clicked Cancel
         if ( result == DialogResult.Cancel )
            return;

         // show error if user specified invalid file
         if ( fileName == "" || fileName == null )
            MessageBox.Show("Invalid File Name", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error);
         else
         {
            // open file if file already exists
            try
            {
               // create FileStream to hold records
               fileOutput = new FileStream( fileName,
                  FileMode.Open, FileAccess.Write );

               // create object for writing bytes to file
               binaryOutput = new BinaryWriter( fileOutput );

               // disable Open button and enable Enter button
               openButton.Enabled = false;
               enterButton.Enabled = true;
            }

```

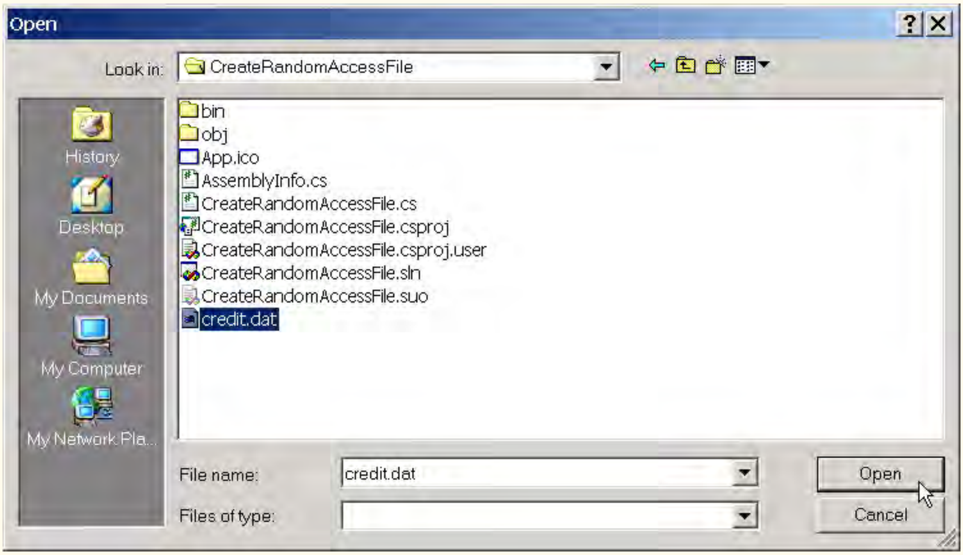
Fig. 17.16 Writing records to random-access files. (Part 2 of 5.)

```
         // notify user if file does not exist
         catch( IOException )
         {
            MessageBox.Show("File Does Not Exits", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error);
         }
      }

   } // end method openButton_Click

   // invoked when user clicks Enter button
   private void enterButton_Click(
      object sender, System.EventArgs e )
   {
      // TextBox values string array
      string[] values = GetTextBoxValues();

      // determine whether TextBox account field is empty
      if ( values[ ( int )TextBoxIndices.ACCOUNT ] != "" )
      {
         // write record to file at appropriate position
         try
         {
            // get account number value from TextBox
            int accountNumber = Int32.Parse(
               values[ ( int )TextBoxIndices.ACCOUNT ] );

            // determine whether accountNumber is valid
            if ( accountNumber > 0 &&
               accountNumber <= NUMBER_OF_RECORDS )
            {
               // move file position pointer
               fileOutput.Seek( ( accountNumber - 1 ) *
                  RandomAccessRecord.SIZE, SeekOrigin.Begin );

               // write data to file
               binaryOutput.Write( accountNumber );
               binaryOutput.Write(
                  values[ ( int )TextBoxIndices.FIRST ] );
               binaryOutput.Write(
                  values[ ( int )TextBoxIndices.LAST ] );
               binaryOutput.Write( Double.Parse( values[
                  ( int )TextBoxIndices.BALANCE ] ) );
            }
            else
            {
               // notify user if invalid account number
               MessageBox.Show("Invalid Account Number", "Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
         }

```

Fig. 17.16 Writing records to random-access files. (Part 3 of 5.)

```
         // handle number-format exception
         catch( FormatException )
         {
            // notify user if error occurs when formatting numbers
            MessageBox.Show( "Invalid Balance", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }

      ClearTextBoxes(); // clear text box values

   } // end method enterButton_Click

} // end class WriteRandomAccessFileForm
```

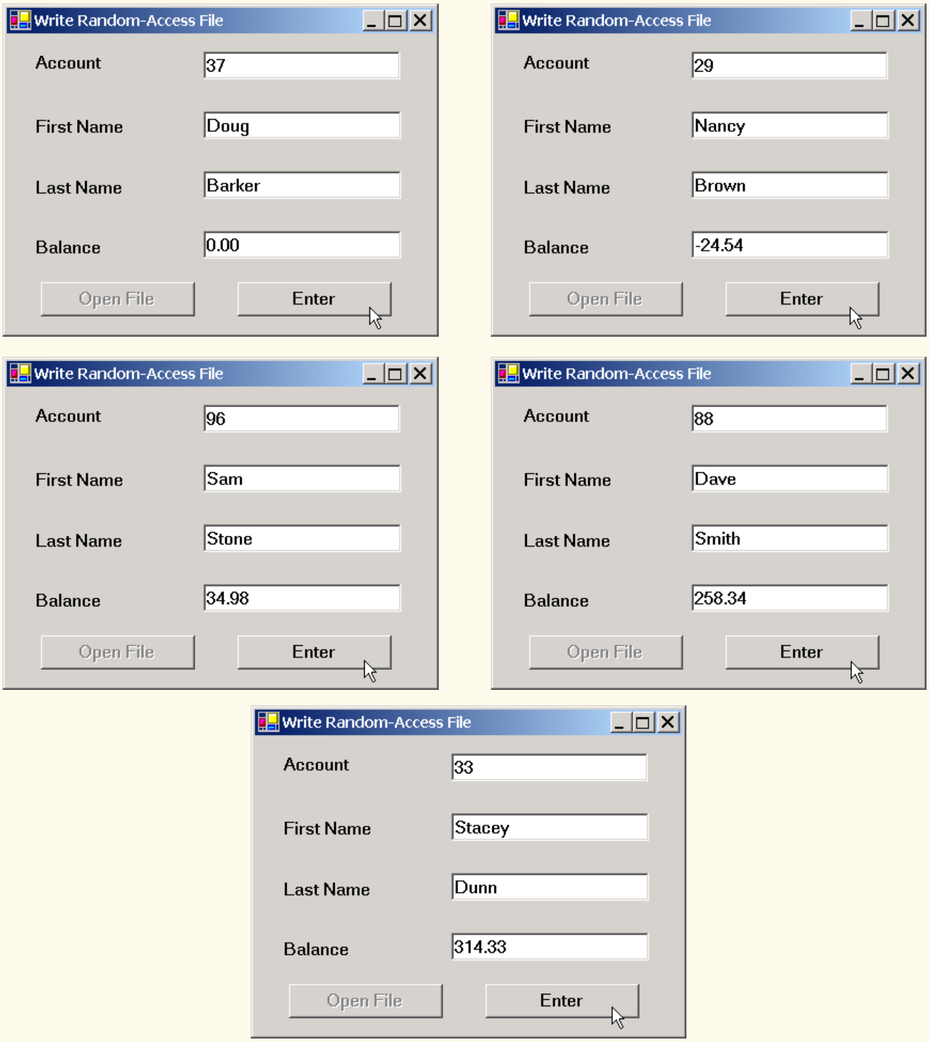

Write Random-Access File
Account
First Name
Last Name
Balance
Open File
Enter


Open
?
History
Desktop
My Documents
My Computer
My Network Pla...
bin
obj
App.ico
AssemblyInfo.cs
CreateRandomAccessFile.cs
CreateRandomAccessFile.csproj
CreateRandomAccessFile.csproj.user
CreateRandomAccessFile.sln
CreateRandomAccessFile.suo
credit.dat
Files of type:
Open
Cancel

Look in: CreateRandomAccessFile

File name: credit.dat

Fig. 17.16 Writing records to random-access files. (Part 4 of 5.)



Account 37
First Name Doug
Last Name Barker
Balance 0.00

Account 29
First Name Nancy
Last Name Brown
Balance -24.54

Account 96
First Name Sam
Last Name Stone
Balance 34.98

Account 88
First Name Dave
Last Name Smith
Balance 258.34

Account 33
First Name Stacey
Last Name Dunn
Balance 314.33

Fig. 17.16 Writing records to random-access files. (Part 5 of 5.)

Class `WriteRandomAccessFileForm` must determine the location in the `FileStream` at which to insert the data from the `TextBox`es. Lines 108–109 use method `Seek` of the `FileStream` object to locate an exact point in the file. In this case, method `Seek` sets the position of the file-position pointer for the `FileStream` object to the byte location calculated by `(accountNumber - 1) * RandomAccessRecord.SIZE`. Because the account numbers range from `1` to `100`, we subtract `1` from the account number when calculating the byte location of the record. For example, our use of method `Seek` sets the first record's file-position pointer to byte 0 of the file (the file's beginning). The second argument to method `Seek` is a member of the enumeration ***`SeekOrigin`*** and specifies the location at which the method should begin seeking. We use `const` ***`SeekOrigin.Begin`***, because we want the method to seek in relation to the beginning of the file.

After the program determines the file location at which to place the record, lines 112–118 write the record to the file using the **`BinaryWriter`** (discussed in the previous section).

17.10 Reading Data Sequentially from a Random-Access File

In the previous sections, we created a random-access file and wrote data to that file. Here, we develop a program (Fig. 17.17) that opens the file, reads records from it and displays only the records that contain data (i.e., those records in which the account number is not zero). This program also provides an additional benefit. Students should attempt to determine what it is—we will reveal it at the end of this section.

When the user clicks the **Open File** button, class **`ReadRandomAccessFileForm`** invokes method **`openButton_Click`** (lines 41–75), which displays the **`OpenFileDialog`** for specifying the file from which to read data. Lines 62–63 instantiate a **`FileStream`** object that opens a file with read-only access. Line 66 creates an instance of class ***`BinaryReader`***, which reads bytes from a stream. We pass the **`FileStream`** object as an argument to the **`BinaryReader`** constructor, thus enabling the **`BinaryReader`** to read bytes from the file.

```
1   // Fig 17.17: ReadRandomAccessFile.cs
2   // Reads and displays random-access file contents.
3
4   // C# namespaces
5   using System;
6   using System.Drawing;
7   using System.Collections;
8   using System.ComponentModel;
9   using System.Windows.Forms;
10  using System.Data;
11  using System.IO;
12
13  // Deitel namespaces
14  using BankLibrary;
15
16  public class ReadRandomAccessFileForm : BankUIForm
17  {
18     private System.Windows.Forms.Button openButton;
19     private System.Windows.Forms.Button nextButton;
20
21     private System.ComponentModel.Container components = null;
22
23     // stream through which data are read from file
24     private FileStream fileInput;
25
26     // stream for reading bytes from file
27     private BinaryReader binaryInput;
28
29     // index of current record to be displayed
30     private int currentRecordIndex;
31
```

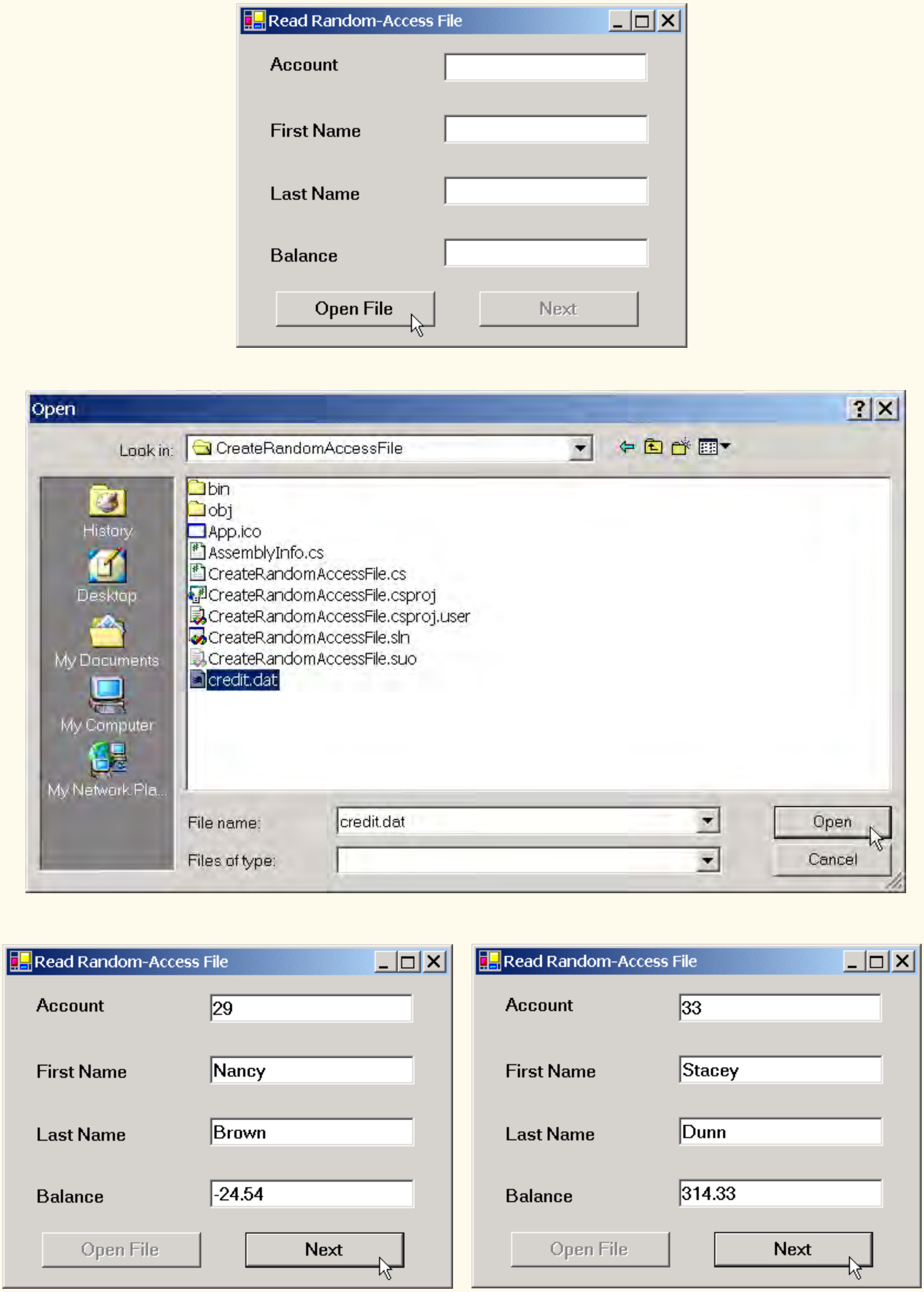
Fig. 17.17 Reading records from random-access files sequentially. (Part 1 of 5.)

```
      [STAThread]
      static void Main()
      {
         Application.Run( new ReadRandomAccessFileForm() );
      }

      // Visual Studio .NET generated code

      // invoked when user clicks Open button
      private void openButton_Click(
         object sender, System.EventArgs e )
      {
         // create dialog box enabling user to open file
         OpenFileDialog fileChooser = new OpenFileDialog();
         DialogResult result = fileChooser.ShowDialog();

         // get file name from user
         string fileName = fileChooser.FileName;

         // exit eventhandler if user clicked Cancel
         if ( result == DialogResult.Cancel )
            return;

         // show error if user specified invalid file
         if ( fileName == "" || fileName == null )
            MessageBox.Show( "Invalid File Name", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         else
         {
            // create FileStream to obtain read access to file
            fileInput = new FileStream( fileName,
               FileMode.Open, FileAccess.Read );

            // use FileStream for BinaryWriter to read bytes from file
            binaryInput = new BinaryReader( fileInput );

            openButton.Enabled = false; // disable Open button
            nextButton.Enabled = true; // enable Next button

            currentRecordIndex = 0;
            ClearTextBoxes();
         }

      } // end method openButton_Click

      // invoked when user clicks Next button
      private void nextButton_Click(
         object sender, System.EventArgs e )
      {
         // record to store file data
         RandomAccessRecord record = new RandomAccessRecord();

```

Fig. 17.17 Reading records from random-access files sequentially. (Part 2 of 5.)

```
         // read record and store data in TextBoxes
         try
         {
            string[] values; // for storing TextBox values

            // get next record available in file
            while( record.Account == 0 )
            {
               // set file position pointer to next record in file
               fileInput.Seek(
                  currentRecordIndex * RandomAccessRecord.SIZE, 0 );

               currentRecordIndex += 1;

               // read data from record
               record.Account = binaryInput.ReadInt32();
               record.FirstName = binaryInput.ReadString();
               record.LastName = binaryInput.ReadString();
               record.Balance = binaryInput.ReadDouble();
            }

            // store record values in temporary string array
            values = new string[] {
               record.Account.ToString(),
               record.FirstName,
               record.LastName,
               record.Balance.ToString() };

            // copy string array values to TextBox values
            SetTextBoxValues( values );
         }

         // handle exception when no records in file
         catch( IOException )
         {
            // close streams if no records in file
            fileInput.Close();
            binaryInput.Close();

            openButton.Enabled = true; // enable Open button
            nextButton.Enabled = false; // disable Next button
            ClearTextBoxes();

            // notify user if no records in file
            MessageBox.Show("No more records in file", "",
               MessageBoxButtons.OK, MessageBoxIcon.Information);
         }

      } // end method nextButton_Click

   } // end class ReadRandomAccessFileForm
```

Fig. 17.17 Reading records from random-access files sequentially. (Part 3 of 5.)



Read Random-Access File

Account

First Name

Last Name

Balance

Open File
Next

Open

Look in: CreateRandomAccessFile

bin
obj
App.ico
AssemblyInfo.cs
CreateRandomAccessFile.cs
CreateRandomAccessFile.csproj
CreateRandomAccessFile.csproj.user
CreateRandomAccessFile.sln
CreateRandomAccessFile.suo
credit.dat

File name: credit.dat

Files of type:

History
Desktop
My Documents
My Computer
My Network Pla...
Open
Cancel

Read Random-Access File

Account: 29
First Name: Nancy
Last Name: Brown
Balance: -24.54

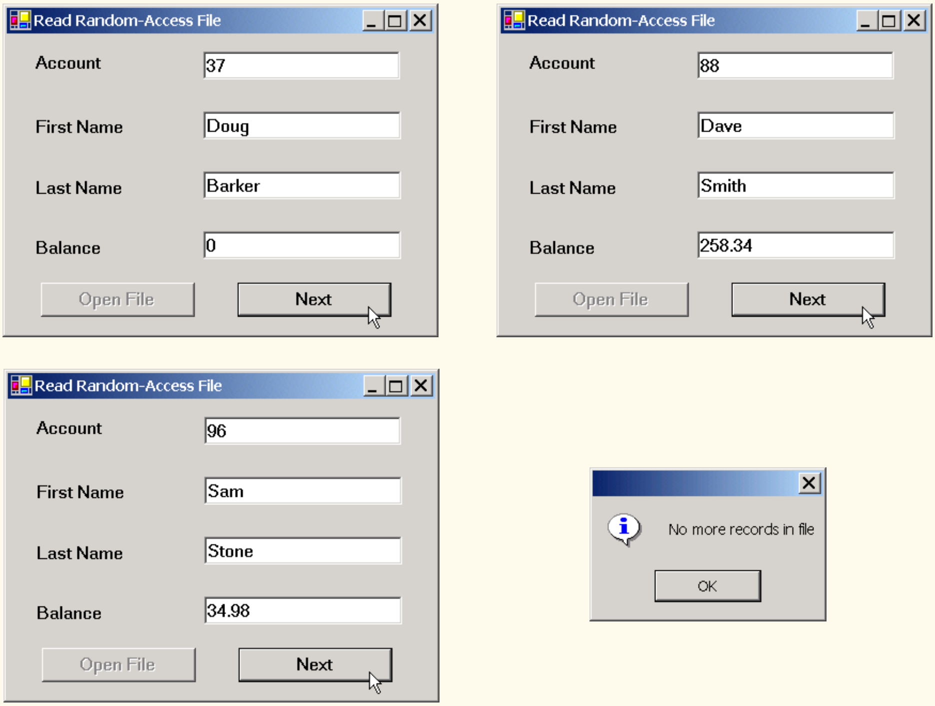
Open File
Next

Read Random-Access File

Account: 33
First Name: Stacey
Last Name: Dunn
Balance: 314.33

Open File
Next

Fig. 17.17 Reading records from random-access files sequentially. (Part 4 of 5.)



Account 37
First Name Doug
Last Name Barker
Balance 0

Account 88
First Name Dave
Last Name Smith
Balance 258.34

Account 96
First Name Sam
Last Name Stone
Balance 34.98

No more records in file

Fig. 17.17 Reading records from random-access files sequentially. (Part 5 of 5.)

When the user clicks the **Next** button, the program calls method `nextButton_Click` (lines 78–132), which reads the next record from the file. Line 82 instantiates a `RandomAccessRecord` for storing the record data from the file. Lines 90–114 define a `while` loop that reads from the file until it reaches a record that has a non-zero account number (`0` is the initial value for the account number). Lines 93–94 call method `Seek` of the `FileStream` object, which moves the file-position pointer to the appropriate place in the file where the record must be read. To accomplish this, method `Seek` uses `int currentRecordIndex`, which stores the number of records that have been read. Lines 99–102 use the `BinaryReader` object to store the file data in the `RandomAccessRecord` object. Recall that class `BinaryWriter` provides overloaded `Write` methods for writing data. However, class `BinaryReader` does not provide overloaded `Read` methods to read data. This means that we must use method `ReadInt32` to read an `int`, method `ReadString` to read a `string` and method `ReadDouble` to read a `double`. Note that the order of these method invocations must correspond to the order in which the `BinaryWriter` object wrote each data type. When the `BinaryReader` reads a valid account number (i.e., a non-zero value), the loop terminates, and lines 106–113 display the record values in the `TextBox`es. When the program has displayed all records, method `Seek` throws an `IOException` (because method `Seek` tries to position the file-position pointer to a location that is beyond the end-of-file marker). The `catch` block (lines 117–130) handles this exception by closing the `FileStream` and `BinaryReader` objects (lines 120–121) and notifying the user that no more records exist (lines 128–129).

What about that additional benefit we promised? If students examine the GUI as the program executes, they will notice that the program displays the records in ascending order by account number! This is a simple consequence of using our direct-access techniques to store these records in the file. Sorting with direct-access techniques is much faster than sorting with the bubble sort presented in Chapter 7, Arrays. We achieve this improved speed by making the file large enough to hold every possible record that a user might create. Of course, this means that the file could be sparsely occupied most of the time, resulting in a waste of storage. Here is yet another example of the space/time trade-off: By using large amounts of space, we are able to develop a faster sorting algorithm.

17.11 Case Study: A Transaction-Processing Program

We now develop a substantial transaction-processing program (Fig. 17.18–Fig. 17.23) using a random-access file to achieve "instant-access" processing. The program maintains a bank's account information. Users of this program can add new accounts, update existing accounts and delete accounts that are no longer needed. First, we discuss the transaction-processing behavior (i.e., the class that enables the addition, updating and removal of accounts). We then discuss the GUI, which contains windows that display the account information and enable the user to invoke the application's transaction-processing behavior.

Transaction-Processing Behavior

In this case study, we create class **`Transaction`** (Fig. 17.18), which acts as a *proxy* to handle all transaction processing. Rather than providing the transaction-processing behavior themselves, the objects in this application use an instance of **`Transaction`** to provide the necessary functionality. By using a proxy, we can encapsulate transaction-processing behavior in only one class, enabling various other classes in our application to reuse this behavior. Furthermore, if we decide to modify this behavior, we modify only the proxy (i.e., class **`Transaction`**), instead of having to modify the behavior of each class that uses the proxy.

Class **`Transaction`** contains methods **`OpenFile`**, **`GetRecord`** and **`AddRecord`**. Method **`OpenFile`** (lines 27–74) uses constant **`FileMode.OpenOrCreate`** (line 33) to create a **`FileStream`** object from either an existing file or one not yet created. Lines 36–39 use this **`FileStream`** to create **`BinaryReader`** and **`BinaryWriter`** objects for reading and writing bytes to the file, respectively. If the file is new, lines 42–64 populate the **`FileStream`** object with empty records. Students might recall that we used these techniques in Section 17.8.

```
1  // Fig. 17.18: Transaction.cs
2  // Handles record transactions.
3
4  // C# namespaces
5  using System;
6  using System.IO;
7  using System.Windows.Forms;
8
```

Fig. 17.18 Record-transaction class for the transaction-processor case study. (Part 1 of 4.)

```
// Deitel namespaces
using BankLibrary;

public class Transaction
{
   // number of records to write to disk
   private const int NUMBER_OF_RECORDS = 100;

   // stream through which data move to and from file
   private FileStream file;

   // stream for reading bytes from file
   private BinaryReader binaryInput;

   // stream for writing bytes to file
   private BinaryWriter binaryOutput;

   // create/open file containing empty records
   public void OpenFile( string fileName )
   {
      // write empty records to file
      try
      {
         // create FileStream from new file or existing file
         file = new FileStream( fileName, FileMode.OpenOrCreate );

         // use FileStream for BinaryWriter to read bytes from file
         binaryInput = new BinaryReader( file );

         // use FileStream for BinaryWriter to write bytes to file
         binaryOutput = new BinaryWriter( file );

         // determine whether file has just been created
         if ( file.Length == 0 )
         {
            // record to be written to file
            RandomAccessRecord blankRecord =
               new RandomAccessRecord();

            // new record can hold NUMBER_OF_RECORDS records
            file.SetLength( RandomAccessRecord.SIZE *
                NUMBER_OF_RECORDS );

            // write blank records to file
            for ( int i = 0; i < NUMBER_OF_RECORDS; i++ )
            {
               // move file-position pointer to next position
               file.Position = i * RandomAccessRecord.SIZE;

               // write blank record to file
               binaryOutput.Write( blankRecord.Account );
               binaryOutput.Write( blankRecord.FirstName );
```

Fig. 17.18 Record-transaction class for the transaction-processor case study. (Part 2 of 4.)

```
                  binaryOutput.Write( blankRecord.LastName );
                  binaryOutput.Write( blankRecord.Balance );
               }
            }
         }

         // notify user of error during writing of blank records
         catch( IOException )
         {
            MessageBox.Show("Cannot create file", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error);
         }

      } // end method OpenFile

      // retrieve record depending on whether account is valid
      public RandomAccessRecord GetRecord( string accountValue )
      {
         // store file data associated with account in record
         try
         {
            // record to store file data
            RandomAccessRecord record = new RandomAccessRecord();

            // get value from TextBox's account field
            int accountNumber = Int32.Parse( accountValue );

            // if account is invalid, do not read data
            if ( accountNumber < 1 ||
               accountNumber > NUMBER_OF_RECORDS )
            {
               // set record's account field with account number
               record.Account = accountNumber;
            }

            // get data from file if account is valid
            else
            {
               // locate position in file where record exists
               file.Seek( ( accountNumber - 1 ) *
                   RandomAccessRecord.SIZE, 0 );

               // read data from record
               record.Account = binaryInput.ReadInt32();
               record.FirstName = binaryInput.ReadString();
               record.LastName = binaryInput.ReadString();
               record.Balance = binaryInput.ReadDouble();
            }

            return record;
         }

```

Fig. 17.18 Record-transaction class for the transaction-processor case study. (Part 3 of 4.)

```
113        // notify user of error during reading
114        catch( IOException )
115        {
116           MessageBox.Show( "Cannot read file", "Error",
117              MessageBoxButtons.OK, MessageBoxIcon.Error );
118        }
119
120        return null;
121
122     } // end method GetRecord;
123
124     // add record to file at position determined by accountNumber
125     public bool AddRecord(
126        RandomAccessRecord record, int accountNumber )
127     {
128        // write record to file
129        try
130        {
131           // move file position pointer to appropriate position
132           file.Seek( ( accountNumber - 1 ) *
133              RandomAccessRecord.SIZE, 0 );
134
135           // write data to file
136           binaryOutput.Write(record.Account);
137           binaryOutput.Write(record.FirstName);
138           binaryOutput.Write(record.LastName);
139           binaryOutput.Write(record.Balance);
140        }
141
142        // notify user if error occurs during writing
143        catch( IOException )
144        {
145           MessageBox.Show( "Error Writing To File", "Error",
146              MessageBoxButtons.OK, MessageBoxIcon.Error );
147
148           return false; // failure
149        }
150
151        return true; // success
152
153     } // end method AddRecord
154
155  } // end class Transaction
```

Fig. 17.18 Record-transaction class for the transaction-processor case study. (Part 4 of 4.)

Method **`GetRecord`** (lines 77–122) returns the record associated with the account-number parameter. Line 83 instantiates a **`RandomAccessRecord`** object that will store the file data. If the account parameter is valid, lines 100–101 call method **`Seek`** of the **`FileStream`** object, which uses the parameter to determine the position of the specified record in the file. Lines 104–107 then call methods **`ReadInt32`**, **`ReadString`** and **`ReadDouble`** of the **`BinaryReader`** object to store the file data in the **`Random-`**

AccessRecord object. Line 110 returns the **RandomAccessRecord** object. We used these techniques in Section 17.10.

Method **AddRecord** (lines 125–153) inserts a record into the file. Lines 132–133 call method **Seek** of the **FileStream** object, which uses the account-number parameter to locate the position at which to insert the record in the file. Lines 136–139 call the overloaded **Write** methods of the **BinaryWriter** object to write the **RandomAccessRecord** object's data to the file. We used these techniques in Section 17.9. Note that, if an error occurs when adding the record (i.e., either the **FileStream** or the **BinaryWriter** throws an **IOException**), lines 145–146 notify the user of the error and return **false** (failure).

Transaction-Processor GUI

The GUI for this program uses a multiple-document interface. Class **TransactionProcessorForm** (Fig. 17.19) is the parent window, and contains corresponding child windows **StartDialogForm** (Fig. 17.20), **NewDialogForm** (Fig. 17.22), **UpdateDialogForm** (Fig. 17.21) and **DeleteDialogForm** (Fig. 17.23). **StartDialogForm** allows the user to open a file containing account information and provides access to the **NewDialogForm**, **UpdateDialogForm** and **DeleteDialogForm** internal frames. These frames allow users to update, create and delete records, respectively.

Initially, **TransactionProcessorForm** displays the **StartDialogForm** object; this window provides the user with various options. It contains four buttons, which enable the user to create or open a file, create a record, update an existing record or delete an existing record.

```
// Fig. 17.19: TransactionProcessor.cs
// MDI parent for transaction-processor application.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class TransactionProcessorForm
   : System.Windows.Forms.Form
{
   private System.ComponentModel.Container components = null;
   private System.Windows.Forms.MdiClient MdiClient1;

   // reference to StartDialog
   private StartDialogForm startDialog;

   // constructor
   public TransactionProcessorForm()
   {
      // required for Windows Form Designer support
      InitializeComponent();

```

Fig. 17.19 **TransactionProcessorForm** class runs the transaction-processor application. (Part 1 of 2.)

```
26          startDialog = new StartDialogForm();
27          startDialog.MdiParent = this;
28          startDialog.Show();
29       }
30
31       [STAThread]
32       static void Main()
33       {
34          Application.Run( new TransactionProcessorForm() );
35       }
36
37       // Visual Studio .NET generated code
38
39    } // end class TransactionProcessorForm
```

Fig. 17.19 **`TransactionProcessorForm`** class runs the transaction-processor application. (Part 2 of 2.)

Before the user can modify records, the user must either create or open a file. When the user clicks the **New/Open File** button, the program calls method **`openButton_Click`** (lines 42–100), which opens a file that the application uses for modifying records. Lines 46–62 display the **`OpenFileDialog`** for specifying the file from which to read data, then use this file to create the **`FileStream`** object. Note that line 52 sets property **`CheckFileExists`** of the **`OpenFileDialog`** object to **`false`**—this enables the user to create a file if the specified file does not exist. If this property were **`true`** (its default value), the dialog would notify the user that the specified file does not exist, thus preventing the user from creating a file.

```
1  // Fig. 17.20: StartDialog.cs
2  // Initial dialog box displayed to user. Provides buttons for
3  // creating/opening file and for adding, updating and removing
4  // records from file.
5
6  // C# namespaces
7  using System;
8  using System.Drawing;
9  using System.Collections;
10 using System.ComponentModel;
11 using System.Windows.Forms;
12
13 // Deitel namespaces
14 using BankLibrary;
15
16 public delegate void MyDelegate();
17
18 public class StartDialogForm : System.Windows.Forms.Form
19 {
20    private System.Windows.Forms.Button updateButton;
21    private System.Windows.Forms.Button newButton;
```

Fig. 17.20 **`StartDialogForm`** class enables users to access dialog boxes associated with various transactions. (Part 1 of 4.)

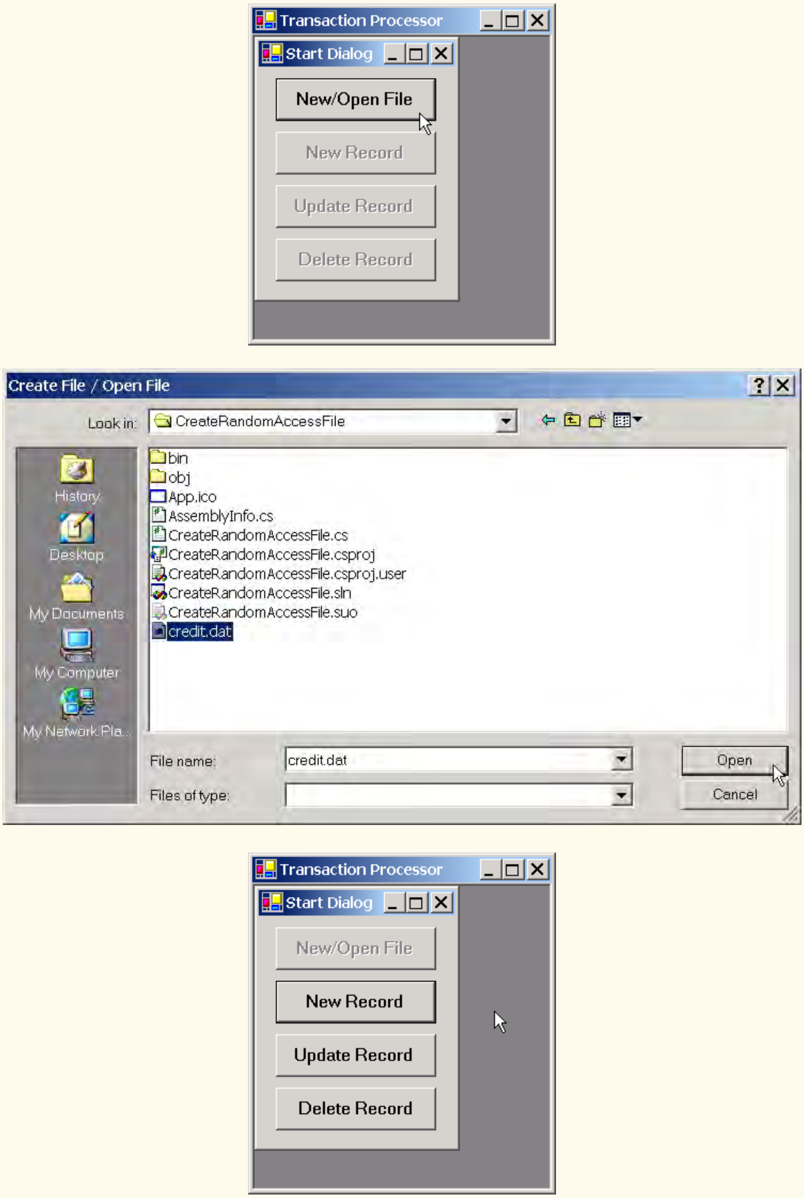
```
      private System.Windows.Forms.Button deleteButton;
      private System.Windows.Forms.Button openButton;

      private System.ComponentModel.Container components = null;

      // reference to dialog box for adding record
      private NewDialogForm newDialog;

      // reference to dialog box for updating record
      private UpdateDialogForm updateDialog;

      // reference to dialog box for removing record
      private DeleteDialogForm deleteDialog;

      // reference to object that handles transactions
      private Transaction transactionProxy;

      // Visual Studio .NET generated code

      // invoked when user clicks New/Open File button
      private void openButton_Click(
         object sender, System.EventArgs e )
      {
         // create dialog box enabling user to create or open file
         OpenFileDialog fileChooser = new OpenFileDialog();
         DialogResult result;
         string fileName;

         // enable user to create file if file does not exist
         fileChooser.Title = "Create File / Open File";
         fileChooser.CheckFileExists = false;

         // show dialog box to user
         result = fileChooser.ShowDialog();

         // exit event handler if user clicked Cancel
         if ( result == DialogResult.Cancel )
            return;

         // get file name from user
         fileName = fileChooser.FileName;

         // show error if user specified invalid file
         if ( fileName == "" || fileName == null )
            MessageBox.Show( "Invalid File Name", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );

         // open or create file if user specified valid file
         else
         {
            // create Transaction with specified file
            transactionProxy = new Transaction();
```

Fig. 17.20 `StartDialogForm` class enables users to access dialog boxes associated with various transactions. (Part 2 of 4.)

```
         transactionProxy.OpenFile( fileName );

         // enable GUI buttons except for New/Open File button
         newButton.Enabled = true;
         updateButton.Enabled = true;
         deleteButton.Enabled = true;
         openButton.Enabled = false;

         // instantiate dialog box for creating records
         newDialog = new NewDialogForm( transactionProxy,
            new MyDelegate( ShowStartDialog ) );

         // instantiate dialog box for updating records
         updateDialog = new UpdateDialogForm( transactionProxy,
            new MyDelegate( ShowStartDialog ) );

         // instantiate dialog box for removing records
         deleteDialog = new DeleteDialogForm( transactionProxy,
            new MyDelegate( ShowStartDialog ) );

         // set StartDialog as MdiParent for dialog boxes
         newDialog.MdiParent = this.MdiParent;
         updateDialog.MdiParent = this.MdiParent;
         deleteDialog.MdiParent = this.MdiParent;
      }

   } // end method openButton_Click

   // invoked when user clicks New Record button
   private void newButton_Click(
      object sender, System.EventArgs e )
   {
      Hide(); // hide StartDialog
      newDialog.Show(); // show NewDialog

   } // end method newButton_Click

   private void updateButton_Click(
      object sender, System.EventArgs e )
   {
      Hide(); // hide StartDialog
      updateDialog.Show(); // show UpdateDialog

   } // end method updateButton_Click

   private void deleteButton_Click(
      object sender, System.EventArgs e )
   {
      Hide(); // hide StartDialog
      deleteDialog.Show(); // show DeleteDialog

   } // end method deleteButton_Click
```

Fig. 17.20 `StartDialogForm` class enables users to access dialog boxes associated with various transactions. (Part 3 of 4.)

```

   protected void ShowStartDialog()
   {
      Show();
   }

} // end class StartDialogForm
```


Transaction Processor
Start Dialog
New/Open File
New Record
Update Record
Delete Record
Create File / Open File
bin
obj
App.ico
AssemblyInfo.cs
CreateRandomAccessFile.cs
CreateRandomAccessFile.csproj
CreateRandomAccessFile.csproj.user
CreateRandomAccessFile.sln
CreateRandomAccessFile.suo
credit.dat
History
Desktop
My Documents
My Computer
My Network Pla...
Files of type:
Open
Cancel
Transaction Processor
Start Dialog
New/Open File
New Record
Update Record
Delete Record

Look in: CreateRandomAccessFile

File name: credit.dat

Fig. 17.20 **`StartDialogForm`** class enables users to access dialog boxes associated with various transactions. (Part 4 of 4.)

If the user specifies a file name, line 73 instantiates an object of class **`Transaction`** (Fig. 17.18), which acts as the proxy for creating, reading records from and writing records to random-access files. Line 74 calls **`Transaction`**'s method **`OpenFile`**, which either creates or opens the specified file, depending on whether the file exists.

Class **`StartDialogForm`** also creates internal windows that enable the user to create, update and delete records. We do not use the default constructor created by Visual Studio .NET for these classes; instead, we use an overloaded constructor that takes as arguments the **`Transaction`** object and a delegate object that references method **`ShowStartDialog`** (lines 127–130). Each child window uses the second delegate parameter to display the **`StartDialogForm`** GUI when the user closes a child window. Lines 83–92 instantiate objects of classes **`UpdateDialogForm`**, **`NewDialogForm`** and **`DeleteDialogForm`**, which serve as the child windows.

When the user clicks the **New Record** button in the **Start Dialog**, the program invokes method **`newButton_Click`** of class **`StartDialogForm`** (Fig. 17.20, lines 103–109), which displays the **`NewDialogForm`** internal frame (Fig. 17.22). Class **`NewDialogForm`** enables the user to create records in the file that **`StartDialogForm`** opened (or created). Line 25 of Fig. 17.22 defines **`MyDelegate`** as a delegate to a method that does not return a value and has no parameters; method **`ShowStartDialog`** of class **`StartDialogForm`** (Fig. 17.20, lines 127–130) conforms to these requirements. Class **`NewDialogForm`** receives a **`MyDelegate`** object, which references this method as a parameter—therefore, **`NewDialogForm`** can invoke this method to display the start window when the user exits the **`NewDialogForm`**. Classes **`UpdateDialogForm`** and **`DeleteDialogForm`** also receive **`MyDelegate`** references as arguments, enabling them to display **`StartDialogForm`** after completing their tasks.

After the user enters data in the **`TextBox`**es and clicks the **Save Record** button, the program invokes method **`saveButton_Click`** (lines 51–66) to write the record to disk. Lines 54–56 call method **`GetRecord`** of the **`Transaction`** object, which should return an empty **`RandomAccessRecord`**. If method **`GetRecord`** returns a **`RandomAccessRecord`** that contains data, the user is attempting to overwrite that **`RandomAccessRecord`** with a new one. Line 60 calls **`private`** method **`InsertRecord`** (lines 69–116). If the **`RandomAccessRecord`** is empty, method **`InsertRecord`** calls method **`AddRecord`** of the **`Transaction`** object (lines 100–101), which adds the newly created **`RandomAccessRecord`** to the file. If the user is attempting to overwrite an existing record, lines 81–83 notify the user that the record already exists and return from the method.

When the user clicks the **Update Record** button in the **Start Dialog**, the program invokes method **`updateButton_Click`** of class **`StartDialogForm`** (Fig. 17.20, lines 111–117), which displays the **`UpdateDialogForm`** internal frame (Fig. 17.21). Class **`UpdateDialogForm`** enables the user to update existing records in the file.

To update a record, the user must enter the account number associated with that record. When the user presses *Enter*, **`UpdateDialogForm`** calls method **`accountTextBox_KeyDown`** (lines 45–84) to display the record contents. This method calls method **`GetRecord`** of the **`Transaction`** object (lines 52–54) to retrieve the specified **`RandomAccessRecord`**. If the record is not empty, lines 64–72 populate the **`TextBox`**es with the **`RandomAccessRecord`** values.

The **Transaction** **`TextBox`** initially contains the string **Charge or Payment**. The user should select this text, type the transaction amount (a positive value for a charge or a negative value for a payment), then press *Enter*. The program calls method **`transactionTextBox_KeyDown`** (lines 87–132) to add the user-specified transaction amount to the current balance.

```
 1  // Fig. 17.22: UpdateDialog.cs
 2  // Enables user to update records in file.
 3
 4  // C# namespaces
 5  using System;
 6  using System.Drawing;
 7  using System.Collections;
 8  using System.ComponentModel;
 9  using System.Windows.Forms;
10
11  // Deitel namespaces
12  using BankLibrary;
13
14  public class UpdateDialogForm : BankUIForm
15  {
16     private System.Windows.Forms.Label transactionLabel;
17     private System.Windows.Forms.TextBox transactionTextBox;
18
19     private System.Windows.Forms.Button saveButton;
20     private System.Windows.Forms.Button cancelButton;
21
22     private System.ComponentModel.Container components = null;
23
24     // reference to object that handles transactions
25     private Transaction transactionProxy;
26
27     // delegate for method that displays previous window
28     private MyDelegate showPreviousWindow;
29
30     // initialize components and set members to parameter values
31     public UpdateDialogForm(
32        Transaction transactionProxyValue,
33        MyDelegate delegateValue )
34     {
35        InitializeComponent();
36        showPreviousWindow = delegateValue;
37
38        // instantiate object that handles transactions
39        transactionProxy = transactionProxyValue;
40     }
41
42     // Visual Studio .NET generated code
43
44     // invoked when user enters text in account TextBox
45     private void accountTextBox_KeyDown(
46        object sender, System.Windows.Forms.KeyEventArgs e )
47     {
```

Fig. 17.21 **`UpdateDialogForm`** class enables users to update records in transaction-processor case study. (Part 1 of 5.)

```
      // determine whether user pressed Enter key
      if ( e.KeyCode == Keys.Enter )
      {
         // retrieve record associated with account from file
         RandomAccessRecord record =
            transactionProxy.GetRecord( GetTextBoxValues()
            [ ( int )TextBoxIndices.ACCOUNT ] );

         // return if record does not exist
         if ( record == null )
            return;

         // determine whether record is empty
         if ( record.Account != 0 )
         {
            // store record values in string array
            string[] values = {
               record.Account.ToString(),
               record.FirstName.ToString(),
               record.LastName.ToString(),
               record.Balance.ToString() };

            // copy string array value to TextBox values
            SetTextBoxValues( values );
            transactionTextBox.Text = "[Charge or Payment]";

         }
         else
         {
            // notify user if record does not exist
            MessageBox.Show(
               "Record Does Not Exist", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }

   } // end method accountTextBox_KeyDown

   // invoked when user enters text in transaction TextBox
   private void transactionTextBox_KeyDown(
      object sender, System.Windows.Forms.KeyEventArgs e )
   {
      // determine whether user pressed Enter key
      if ( e.KeyCode == Keys.Enter )
      {
         // calculate balance using transaction TextBox value
         try
         {
            // retrieve record associated with account from file
            RandomAccessRecord record =
               transactionProxy.GetRecord( GetTextBoxValues()
               [ ( int )TextBoxIndices.ACCOUNT ] );
```

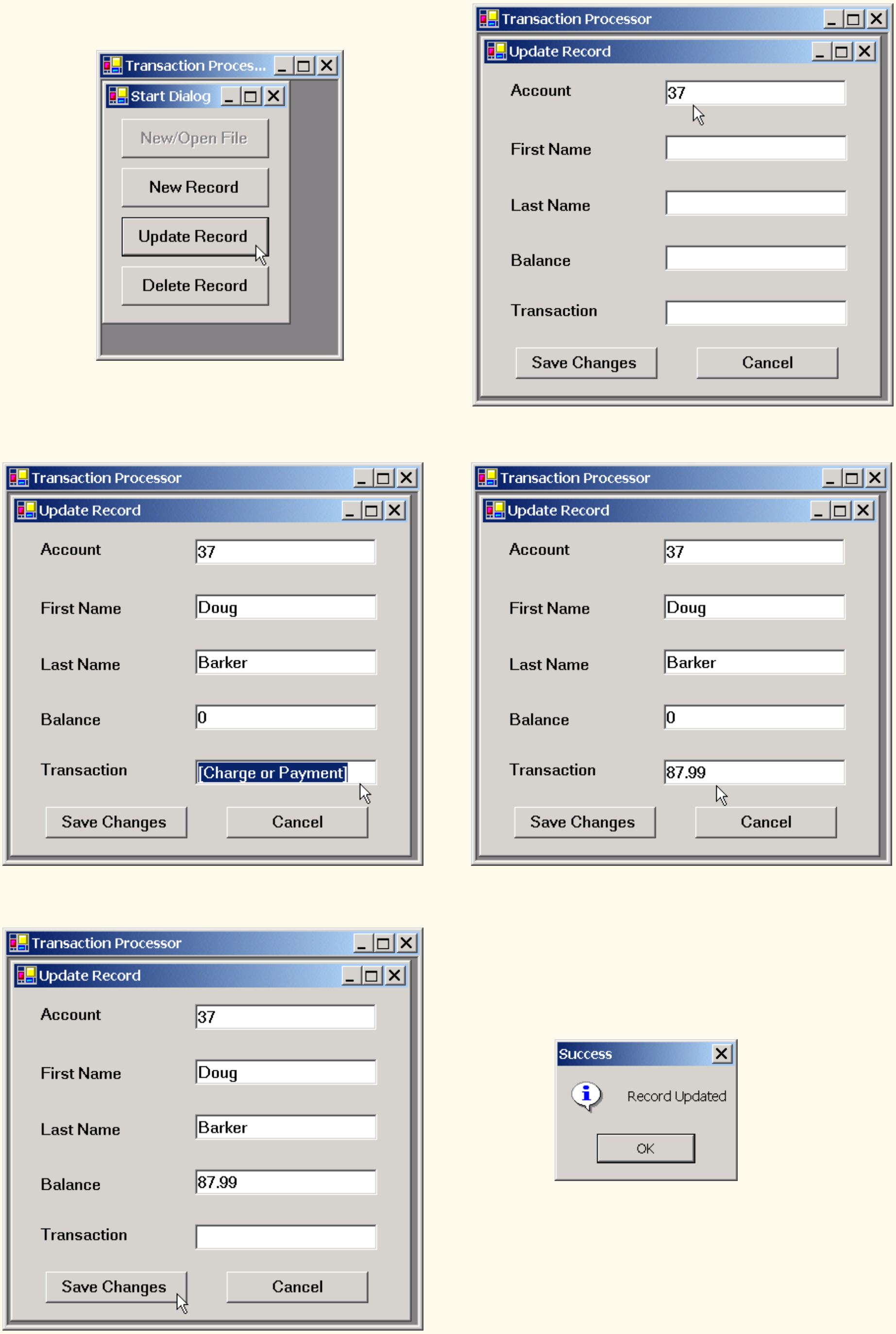
Fig. 17.21 **UpdateDialogForm** class enables users to update records in transaction-processor case study. (Part 2 of 5.)

```

            // get transaction TextBox value
            double transactionValue =
               Double.Parse( transactionTextBox.Text );

            // calculate new balance (old balance + transaction)
            double newBalance =
               record.Balance + transactionValue;

            // store record values in string array
            string[] values = {
               record.Account.ToString(),
               record.FirstName.ToString(),
               record.LastName.ToString(),
               newBalance.ToString() };

            // copy string array value to TextBox values
            SetTextBoxValues( values );

            // clear transaction TextBox
            transactionTextBox.Text = "";
         }

         // notify user if error occurs in parameter mismatch
         catch( FormatException )
         {
            MessageBox.Show(
               "Invalid Transaction", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
         }
      }

   } // end method transactionTextBox_KeyDown

   // invoked when user clicks Save button
   private void saveButton_Click(
      object sender, System.EventArgs e )
   {
      RandomAccessRecord record =
         transactionProxy.GetRecord( GetTextBoxValues()
         [ ( int )TextBoxIndices.ACCOUNT ] );

      // if record exists, update in file
      if ( record != null )
         UpdateRecord( record );

      Hide();
      ClearTextBoxes();
      showPreviousWindow();

   } // end method saveButton_Click

```

Fig. 17.21 **`UpdateDialogForm`** class enables users to update records in transaction-processor case study. (Part 3 of 5.)

```
   // invoked when user clicks Cancel button
   private void cancelButton_Click(
      object sender, System.EventArgs e )
   {
      Hide();
      ClearTextBoxes();
      showPreviousWindow();

   } // end method cancelButton_Click

   // update record in file at position specified by accountNumber
   public void UpdateRecord( RandomAccessRecord record )
   {
      // store TextBox values in record and write record to file
      try
      {
         int accountNumber = record.Account;
         string[] values = GetTextBoxValues();

         // store values in record
         record.Account = accountNumber;
         record.FirstName =
            values[ ( int )TextBoxIndices.FIRST ];
         record.LastName =
            values[ ( int )TextBoxIndices.LAST ];
         record.Balance =
            Double.Parse(
            values[ ( int )TextBoxIndices.BALANCE ] );

         // add record to file
         if ( transactionProxy.AddRecord(
            record, accountNumber ) == false )

            return; // if error
      }

      // notify user if error occurs in parameter mismatch
      catch( FormatException )
      {
         MessageBox.Show( "Invalid Balance", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );

         return;
      }

      MessageBox.Show( "Record Updated", "Success",
         MessageBoxButtons.OK,
         MessageBoxIcon.Information );

   } // end method UpdateRecord

} // end class UpdateDialogForm
```

Fig. 17.21 `UpdateDialogForm` class enables users to update records in transaction-processor case study. (Part 4 of 5.)


Transaction Proces...
Start Dialog
New/Open File
New Record
Update Record
Delete Record
Transaction Processor
Update Record
Save Changes
Cancel
Success
OK

Account: 37
First Name:
Last Name:
Balance:
Transaction:

Account: 37
First Name: Doug
Last Name: Barker
Balance: 0
Transaction: [Charge or Payment]

Account: 37
First Name: Doug
Last Name: Barker
Balance: 0
Transaction: 87.99

Account: 37
First Name: Doug
Last Name: Barker
Balance: 87.99
Transaction:

Record Updated

Fig. 17.21 `UpdateDialogForm` class enables users to update records in transaction-processor case study. (Part 5 of 5.)

The user clicks the **Save Changes** button to write the altered contents of the `TextBox`es to the file. (Note that pressing **Save Changes** does not update the **Balance** field—the user must press *Enter* to update this field before pressing **Save Changes**.) When the user clicks **Save Changes**, the program invokes method `saveButton_Click` (lines 135–150), which calls `private` method `UpdateRecord` (lines 163–201). This method calls method `AddRecord` of the `Transaction` object (lines 182–183) to store the `TextBox` values in a `RandomAccessRecord` and overwrite the existing file record with the `RandomAccessRecord` containing the new data.

When the user clicks the **Delete Record** button of the **Start Dialog**, the program invokes method `deleteButton_Click` of class `StartDialogForm` (Fig. 17.20, lines 119–125), which displays the `DeleteDialogForm` internal frame (Fig. 17.23). Class `DeleteDialogForm` enables the user to remove existing records from the file. To remove a record, the user must enter the account number associated with that record. When the user clicks the **Delete Record** button (now, from the `DeleteDialogForm` internal frame), `DeleteDialogForm` calls method `deleteButton_Click` (lines 44–57). This method calls method `DeleteRecord` (lines 69–102), which ensures that the record to be deleted exists, then calls method `AddRecord` of the `Transaction` object (lines 87–88) to overwrite the file record with an empty one.

In this chapter, we demonstrated how to read data from files and write data to files via both sequential-access and random-access file-processing techniques. Using class `BinaryFormatter`, we serialized and deserialized objects to and from streams; we then employed `FileStream`, `BinaryWriter` and `BinaryReader` to transfer the objects' byte representation to and from files. In Chapter 18, we discuss the Extensible Markup Language (XML), a widely supported technology for describing data. Using XML, we can describe any type of data, such as mathematical formulas, music and financial reports.

```
// Fig. 17.21: NewDialog.cs
// Enables user to insert new record into file.

// C# namespaces
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

// Deitel namespaces
using BankLibrary;

public class NewDialogForm : BankUIForm
{
   private System.Windows.Forms.Button saveButton;
   private System.Windows.Forms.Button cancelButton;
```

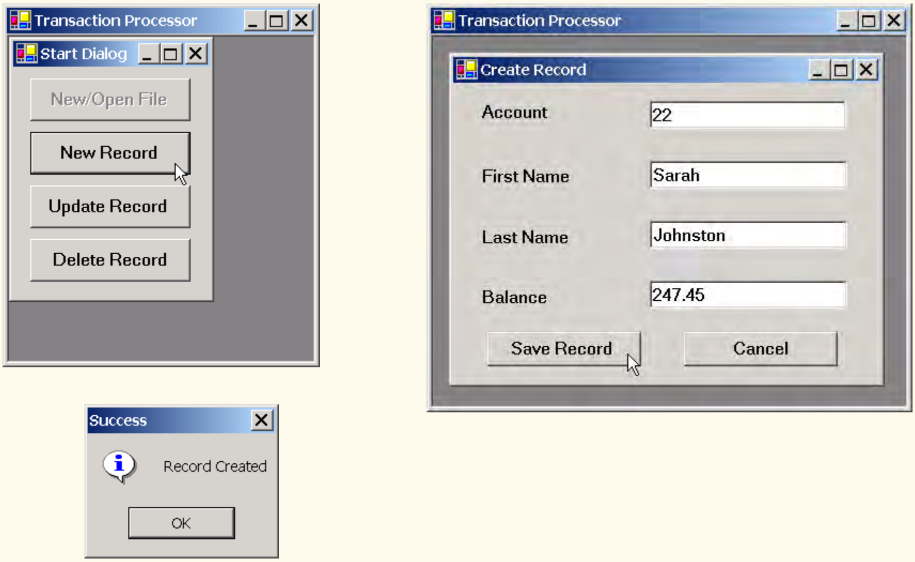
Fig. 17.22 `NewDialogForm` class enables users to create records in transaction-processor case study. (Part 1 of 4.)

```
      private System.ComponentModel.Container components = null;

      // reference to object that handles transactions
      private Transaction transactionProxy;

      // delegate for method that displays previous window
      public MyDelegate showPreviousWindow;

      // constructor
      public NewDialogForm( Transaction transactionProxyValue,
         MyDelegate delegateValue )
      {
         InitializeComponent();
         showPreviousWindow = delegateValue;

         // instantiate object that handles transactions
         transactionProxy = transactionProxyValue;
      }

      // Visual Studio .NET generated code

      // invoked when user clicks Cancel button
      private void cancelButton_Click(
         object sender, System.EventArgs e )
      {
         Hide();
         ClearTextBoxes();
         showPreviousWindow();

      } // end method cancelButton_Click

      // invoked when user clicks Save As button
      private void saveButton_Click(
         object sender, System.EventArgs e )
      {
         RandomAccessRecord record =
            transactionProxy.GetRecord( GetTextBoxValues()
            [ ( int )TextBoxIndices.ACCOUNT ] );

         // if record exists, add it to file
         if ( record != null )
            InsertRecord( record );

         Hide();
         ClearTextBoxes();
         showPreviousWindow();

      } // end method saveButton_Click
```

Fig. 17.22 **`NewDialogForm`** class enables users to create records in transaction-processor case study. (Part 2 of 4.)

```
   // insert record in file at position specified by accountNumber
   private void InsertRecord( RandomAccessRecord record )
   {
      //store TextBox values in string array
      string[] textBoxValues = GetTextBoxValues();

      // store TextBox account field
      int accountNumber = Int32.Parse(
         textBoxValues[ ( int )TextBoxIndices.ACCOUNT ] );

      // notify user and return if record account is not empty
      if ( record.Account != 0 )
      {
         MessageBox.Show(
            "Record Already Exists or Invalid Number", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error);

         return;
      }

      // store values in record
      record.Account = accountNumber;
      record.FirstName =
         textBoxValues[ ( int )TextBoxIndices.FIRST];
      record.LastName =
         textBoxValues[ ( int )TextBoxIndices.LAST];
      record.Balance = Double.Parse(
         textBoxValues[ ( int )TextBoxIndices.BALANCE ] );

      // add record to file
      try
      {
         if ( transactionProxy.AddRecord(
            record, accountNumber ) == false )

            return; // if error
      }

      // notify user if error occurs in parameter mismatch
      catch( FormatException )
      {
         MessageBox.Show( "Invalid Balance", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );
      }

      MessageBox.Show( "Record Created", "Success",
         MessageBoxButtons.OK, MessageBoxIcon.Information );

   } // end method InsertRecord

} // end class NewDialogForm
```

Fig. 17.22 **`NewDialogForm`** class enables users to create records in transaction-processor case study. (Part 3 of 4.)



Transaction Processor
Start Dialog
New/Open File
New Record
Update Record
Delete Record
Success
OK
Transaction Processor
Create Record
Save Record
Cancel

Record Created

Account: 22
First Name: Sarah
Last Name: Johnston
Balance: 247.45

Fig. 17.22 **`NewDialogForm`** class enables users to create records in transaction-processor case study. (Part 4 of 4.)

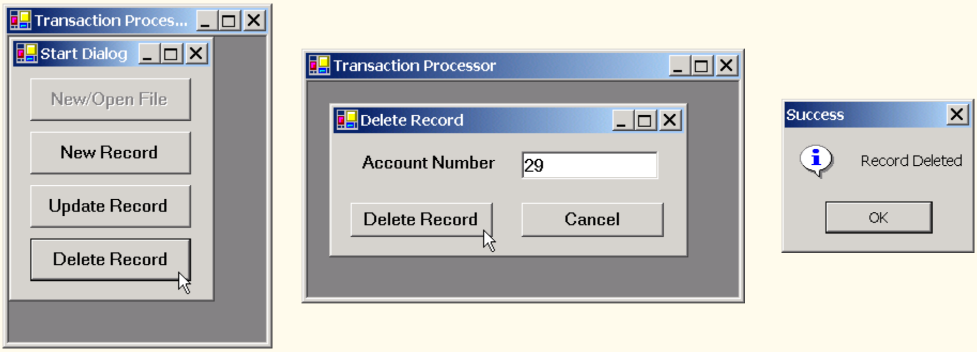
```
// Fig. 17.23: DeleteDialog.cs
// Enables user to delete records in file.

// C# namespaces
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

// Deitel namespaces
using BankLibrary;

public class DeleteDialogForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label accountLabel;
   private System.Windows.Forms.TextBox accountTextBox;

   private System.Windows.Forms.Button deleteButton;
   private System.Windows.Forms.Button cancelButton;

   private System.ComponentModel.Container components = null;

   // reference to object that handles transactions
   private Transaction transactionProxy;
```

Fig. 17.23 **`DeleteDialogForm`** class enables users to remove records from files in transaction-processor case study. (Part 1 of 3.)

```

      // delegate for method that displays previous window
      private MyDelegate showPreviousWindow;

      // initialize components and set members to parameter values
      public DeleteDialogForm( Transaction transactionProxyValue,
         MyDelegate delegateValue)
      {
         InitializeComponent();
         showPreviousWindow = delegateValue;

         // instantiate object that handles transactions
         transactionProxy = transactionProxyValue;
      }

      // Visual Studio .NET generated code

      // invoked when user clicks Delete Record button
      private void deleteButton_Click(
         object sender, System.EventArgs e)
      {
         RandomAccessRecord record =
            transactionProxy.GetRecord( accountTextBox.Text );

         // if record exists, delete it in file
         if ( record != null )
            DeleteRecord( record );

         this.Hide();
         showPreviousWindow();

      } // end method deleteButton_Click

      // invoked when user clicks Cancel button
      private void cancelButton_Click(
         object sender, System.EventArgs e)
      {
         this.Hide();
         showPreviousWindow();

      } // end method cancelButton_Click

      // delete record in file at position specified by accountNumber
      public void DeleteRecord( RandomAccessRecord record )
      {
         int accountNumber = record.Account;

         // display error message if record does not exist
         if ( record.Account == 0 )
         {
            MessageBox.Show( "Record Does Not Exist", "Error",
               MessageBoxButtons.OK, MessageBoxIcon.Error );
```

Fig. 17.23 **`DeleteDialogForm`** class enables users to remove records from files in transaction-processor case study. (Part 2 of 3.)

```
         accountTextBox.Clear();

         return;
      }

      // create blank record
      record = new RandomAccessRecord();

      // write over file record with empty record
      if ( transactionProxy.AddRecord(
         record, accountNumber ) == true )

         // notify user of successful deletion
         MessageBox.Show( "Record Deleted", "Success",
            MessageBoxButtons.OK, MessageBoxIcon.Information );
      else

         // notify user of failure
         MessageBox.Show(
            "Record could not be deleted", "Error",
            MessageBoxButtons.OK, MessageBoxIcon.Error );

      accountTextBox.Clear();

   } // end method DeleteRecord

} // end class DeleteDialogForm
```


Transaction Proces...
Start Dialog
New/Open File
New Record
Update Record
Delete Record
Transaction Processor
Delete Record
Delete Record
Cancel
Success
OK

Account Number 29

Record Deleted

Fig. 17.23 **`DeleteDialogForm`** class enables users to remove records from files in transaction-processor case study. (Part 3 of 3.)

SUMMARY

- All data items processed by a computer ultimately are reduced to combinations of zeros and ones.
- The smallest data items that computers support are called bits and can assume either the value `0` or the value `1`.
- Digits, letters and special symbols are referred to as characters. The set of all characters used to write programs and represent data items on a particular computer is called that computer's char-

acter set. Every character in a computer's character set is represented as a pattern of `1`s and `0`s (characters in C# are Unicode characters, which are composed of 2 bytes).

- At least one field in a record is chosen as a record key, which identifies that record as belonging to a particular person or entity and distinguishes that record from all other records in the file.
- A file is a group of related records.
- Files are used for long-term retention of large amounts of data and can store those data even after the program that created the data terminates.
- Data maintained in files often are called persistent data.
- Class **`File`** enables programs to obtain information about a file.
- Class **`Directory`** enables programs to obtain information about a directory.
- Class **`FileStream`** provides method **`Seek`** for repositioning the file-position pointer (the byte number of the next byte in the file to be read or written) to any position in the file.
- The most common type of file organization is the sequential file, in which records typically are stored in order by the record-key field.
- When a file is opened, an object is created, and a stream is associated with the object.
- C# imposes no structure on files. This means that concepts like that of a "record" do not exist in C#. The programmer must structure each file appropriately to meet the requirements of an application.
- A collection of programs designed to create and manage databases is called a database management system (DBMS).
- C# views each file as a sequential stream of bytes.
- Each file ends in some machine-dependent form of end-of-file marker.
- Objects of classes **`OpenFileDialog`** and **`SaveFileDialog`** are used for selecting files to open and save, respectively. Method **`ShowDialog`** of these classes displays that dialog.
- When displayed, both an **`OpenFileDialog`** and a **`SaveFileDialog`** prevent the user from interacting with any other program window until the dialog is closed. Dialogs that behave in this fashion are called modal dialogs.
- Streams provide communication channels between files and programs.
- To perform file processing in C#, the namespace **`System.IO`** must be referenced. This namespace includes definitions for stream classes such as **`StreamReader`**, **`StreamWriter`** and **`FileStream`**. Files are opened by instantiating objects of these classes.
- To retrieve data sequentially from a file, programs normally start from the beginning of the file, reading all data consecutively until the desired data are found.
- With a sequential-access file, each successive input/output request reads or writes the next consecutive set of data in the file.
- Instant data access is possible with random-access files. A program can access individual records of a random-access file directly (and quickly) without searching through other records. Random-access files sometimes are called direct-access files.
- With a random-access file, each successive input/output request can be directed to any part of the file, which can be any distance from the part of the file referenced in the previous request.
- Programmers can use members of the **`FileAccess`** enumeration to control users' access to files.
- Only classes with the **`Serializable`** attribute can be serialized to and deserialized from files.
- There are a variety of techniques for creating random-access files. Perhaps the simplest involves requiring that all records in a file be of the same fixed length.

- The use of fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record in relation to the beginning of the file
- Data can be inserted into a random-access file without destroying other data in the file. Users can also update or delete previously stored data without rewriting the entire file.
- **BinaryFormatter** uses methods **Serialize** and **Deserialize** to write and to read objects, respectively. Method **Serialize** writes the object's representation to a stream. Method **Deserialize** reads this representation from a stream and reconstructs the original object.
- Methods **Serialize** and **Deserialize** each require a **Stream** object as a parameter, enabling the **BinaryFormatter** to access the correct file.
- Class **BinaryReader** and **BinaryWriter** provide methods for reading and writing bytes to streams, respectively. The **BinaryReader** and **BinaryWriter** constructors receive as arguments references to instances of class **System.IO.Stream**.
- Class **FileStream** inherits from class **Stream**, so we can pass the **FileStream** object as an argument to either the **BinaryReader** or **BinaryWriter** constructor to create an object that can transfer bytes directly to or from a file.
- Random-access file-processing programs rarely write a single field to a file. Normally, they write one object at a time.
- Sorting with direct-access techniques is fast. This speed is achieved by making the file large enough to hold every possible record that might be created. Of course, this means that the file could be sparsely occupied most of the time, possibly wasting memory.

TERMINOLOGY

binary digit (bit)
BinaryFormatter class
BinaryReader class
BinaryWriter class
BufferedStream class
bit manipulation
character
character set
Close method of class **StreamReader**
closing a file
Console class
Copy method of class **File**
Create method of class **File**
CreateDirectory method of class **Directory**
CreateText method of class **File**
data hierarchy
database
database management system (DBMS)
Delete method of class **Directory**
Delete method of class **File**
Deserialize method of class **BinaryFormatter**
direct-access files
Directory class
DirectoryInfo class
end-of-file marker
Error property of class **Console**
escape sequence
Exists method of class **Directory**
field
file
File class
file-processing programs
FileAccess enumeration
file-position pointer
FileStream class
fixed-length records
GetCreationTime method of class **Directory**
GetCreationTime method of class **File**
GetDirectories method of class **Directory**
GetFiles method of class **Directory**
GetLastAccessTime method of class **Directory**
GetLastAccessTime method of class **File**
GetLastWriteTime method of class **Directory**
GetLastWriteTime method of class **File**
In property of class **Console**
”instant-access” application

`IOException`
`MemoryStream` class
modal dialog
`Move` method of class `Directory`
`Move` method of class `File`
`Open` method of class `File`
`OpenFileDialog` class
`OpenRead` method of class `File`
`OpenText` method of class `File`
`OpenWrite` method of class `File`
`Out` property of class `Console`
pattern of `1`s and `0`s
persistent data
random-access file
`Read` method of class `Console`
`ReadDouble` method of class `BinaryReader`
`ReadInt32` method of class `BinaryReader`
`ReadLine` method of class `Console`
`ReadLine` method of class `StreamReader`
`ReadString` method of class `BinaryReader`
record
record key
regular expression
`SaveFileDialog` class
secondary storage devices
`Seek` method of class `FileStream`
`SeekOrigin` enumeration
separation character
sequential-access file
`Serializable` attribute
`SerializationException`
`Serialize` method of class `BinaryFormatter`
`ShowDialog` method of class `OpenFileDialog`
`ShowDialog` method of class `SaveFileDialog`
standard error-stream object
standard input-stream object
standard output-stream object
`Stream` class
stream of bytes
stream processing
`StreamReader` class
`StreamWriter` class
`System.IO` namespace
`System.Runtime.Serialization.Formatters.Binary` namespace
`TextReader` class
`TextWriter` class
transaction-processing system
Windows Control Library project
`Write` method of class `BinaryWriter`
`Write` method of class `Console`
`Write` method of class `StreamWriter`
`WriteLine` method of class `Console`
`WriteLine` method of class `StreamWriter`

SELF-REVIEW EXERCISES

17.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Creating instances of classes `File` and `Directory` is impossible.
b) Typically, a sequential file stores records in order by the record-key field.
c) Class `StreamReader` inherits from class `Stream`.
d) Any class can be serialized to a file.
e) Searching a random-access file sequentially to find a specific record is unnecessary.
f) Method `Seek` of class `FileStream` always seeks relative to the beginning of a file.
g) C# provides class `Record` to store records for random-access file-processing applications.
h) Banking systems, point-of-sale systems and automated-teller machines are types of transaction-processing systems.
i) Classes `StreamReader` and `StreamWriter` are used with sequential-access files.
j) Instantiating objects of type `Stream` is impossible.

17.2 Fill in the blanks in each of the following statements:

a) Ultimately, all data items processed by a computer are reduced to combinations of __________ and __________.
b) The smallest data item a computer can process is called a __________.
c) A __________ is a group of related records.
d) Digits, letters and special symbols are collectively referred to as __________.

e) A group of related files is called a __________.
f) **`StreamReader`** method __________ reads a line of text from a file.
g) **`StreamWriter`** method __________ writes a line of text to a file.
h) Method **`Serialize`** of class **`BinaryFormatter`** takes a(n) __________ and a(n) __________ as arguments.
i) The __________ namespace contains most of C#'s file-processing classes.
j) The __________ namespace contains the **`BinaryFormatter`** class.

ANSWERS TO SELF-REVIEW EXERCISES

17.1 a) True. b) True. c) False. **`StreamReader`** inherits from **`TextReader`**. d) False. Only classes with the **`Serializable`** attribute can be serialized. e) True. f) False. It seeks relative to the **`SeekOrigin`** enumeration member that is passed as one of the arguments. g) False. C# imposes no structure on a file, so the concept of a "record" does not exist. h.) True. i) True. j) True.

17.2 a) **`1`**s, **`0`**s. b) bit. c) file. d) characters. e) database. f) **`ReadLine`**. g) **`WriteLine`**. h) **`Stream`**, **`Object`**. i) **`System.IO`**. j) **`System.Runtime.Serialization.Formatters.Binary`**.

EXERCISES

17.3 Create a program that stores student grades in a text file. The file should contain the name, ID number, class taken and grade of every student. Allow the user to load a grade file and display its contents in a read-only textbox. The entries should be displayed as follows:

```
LastName, FirstName:  ID#  Class  Grade
```

We list some sample data below:

```
Jones, Bob: 1 "Introduction to Computer Science" "A-"

Johnson, Sarah: 2 "Data Structures" "B+"

Smith, Sam: 3 "Data Structures" "C"
```

17.4 Modify the previous program to use objects of a class that can be serialized to and deserialized from a file. Ensure fixed-length records by fixing the lengths of the fields **`LastName`**, **`FirstName`**, **`Class`** and **`Grade`**.

17.5 Extend classes **`StreamReader`** and **`StreamWriter`**. Make the class that derives from **`StreamReader`** have methods **`ReadInteger`**, **`ReadBoolean`** and **`ReadString`**. Make the class that derives from **`StreamWriter`** have methods **`WriteInteger`**, **`WriteBoolean`** and **`WriteString`**. Think about how to design the writing methods so that the reading methods will be able to read what was written. Design **`WriteInteger`** and **`WriteBoolean`** to write **`string`**s of uniform size, so that **`ReadInteger`** and **`ReadBoolean`** can read those values accurately. Make sure **`ReadString`** and **`WriteString`** use the same character(s) to separate **`string`**s.

17.6 Create a program that combines the ideas of Fig. 17.9 and Fig. 17.11 to allow a user to write records to and read records from a file. Add an extra field of type **`bool`** to the record to indicate whether the account has overdraft protection.

17.7 In commercial data processing, it is common to have several files in each application system. In an accounts receivable system, for example, there is generally a master file containing detailed information about each customer, such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and possibly a condensed history of recent purchases and cash payments.

As transactions occur (i.e., sales are made and cash payments arrive in the mail), they are entered into a file. At the end of each business period (i.e., a month for some companies, a week for others and a day in some cases), the file of transactions (**`trans.dat`**) is applied to the master file (**`oldmast.dat`**), thus updating each account's record of purchases and payments. During an updating run, the master file is rewritten as a new file (**`newmast.dat`**), which then is used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period, and, therefore, no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments might have just moved to the community, and the company might not have had a chance to create a master record for this customer.

When a match occurs (i.e., records with the same account number appear on both the master file and the transaction file), add the dollar amount on the transaction file to the current balance on the master file and write the **`newmast.dat`** record. (Assume that purchases are indicated by positive amounts on the transaction file and that payments are indicated by negative amounts.) When there is a master record for a particular account, but no corresponding transaction record, merely write the master record to **`newmast.dat`**. When there is a transaction record, but no corresponding master record, print the message "**`Unmatched transaction record for account number...`**" (fill in the account number from the transaction record).

17.8 You are the owner of a hardware store and need to keep an inventory of the different tools you sell, how many of each are currently in stock and the cost of each. Write a program that initializes the random-access file **`hardware.dat`** to 100 empty records, lets you input data relating to each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update any information in the file. The tool identification number should be the record number. Use the information in Fig. 17.24 to start your file.

Record #	Tool name	Quantity	Price
3	Electric sander	18	35.99
19	Hammer	128	10.00
26	Jig saw	16	14.25
39	Lawn mower	10	79.50
56	Power saw	8	89.99
76	Screwdriver	236	4.99
81	Sledge hammer	32	19.75
88	Wrench	65	6.48

Fig. 17.24 Inventory of a hardware store.

18

Extensible Markup Language (XML)

Objectives

- To be able to mark up data, using XML.
- To understand the concept of an XML namespace.
- To understand the relationship between DTDs, Schemas and XML.
- To be able to create Schemas.
- To be able to create and use simple XSLT documents.
- To be able to transform XML documents into XHTML, using class **`XslTransform`**.
- To become familiar with BizTalk™.

Knowing trees, I understand the meaning of patience.
Knowing grass, I can appreciate persistence.
Hal Borland

Like everything metaphysical, the harmony between thought and reality is to be found in the grammar of the language.
Ludwig Wittgenstein

I played with an idea and grew willful, tossed it into the air; transformed it; let it escape and recaptured it; made it iridescent with fancy, and winged it with paradox.
Oscar Wilde

Outline

18.1 Introduction

18.2 XML Documents

18.3 XML Namespaces

18.4 Document Object Model (DOM)

18.5 Document Type Definitions (DTDs), Schemas and Validation

- **18.5.1 Document Type Definitions**
- **18.5.2 Microsoft XML Schemas**
- **18.5.3 W3C XML Schema**
- **18.5.4 Schema Validation in C#**

18.6 Extensible Stylesheet Language and `XslTransform`

18.7 Microsoft BizTalk™

18.8 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

18.1 Introduction

The *Extensible Markup Language* (XML) was developed in 1996 by the *World Wide Web Consortium's (W3C's) XML Working Group*. XML is a portable, widely supported, *open technology* (i.e., non-proprietary technology) for describing data. XML is becoming the standard for storing data that is exchanged between applications. Using XML, document authors can describe any type of data, including mathematical formulas, software-configuration instructions, music, recipes and financial reports. XML documents are readable by both humans and machines.

The .NET Framework uses XML extensively. The Framework Class Library (FCL) provides an extensive set of XML-related classes. Much of Visual Studio's internal implementation also employs XML. In this chapter, we introduce XML, XML-related technologies and key classes for creating and manipulating XML documents.

18.2 XML Documents

In this section, we present our first XML document, which describes an article (Fig. 18.1). [*Note:* The line numbers shown are not part of the XML document.]

```
<?xml version = "1.0"?>

<!-- Fig. 18.1: article.xml      -->
<!-- Article structured with XML -->

<article>

   <title>Simple XML</title>
```

Fig. 18.1 XML used to mark up an article. (Part 1 of 2.)

```
 9
10     <date>December 6, 2001</date>
11
12     <author>
13        <firstName>John</firstName>
14        <lastName>Doe</lastName>
15     </author>
16
17     <summary>XML is pretty easy.</summary>
18
19     <content>In this chapter, we present a wide variety of examples
20        that use XML.
21     </content>
22
23  </article>
```

Fig. 18.1 XML used to mark up an article. (Part 2 of 2.)

This document begins with an optional *XML declaration* (line 1), which identifies the document as an XML document. The ***`version`*** *information parameter* specifies the version of XML that is used in the document. XML comments (lines 3–4), which begin with **`<!--`** and end with **`-->`**, can be placed almost anywhere in an XML document. As in a C# program, comments are used in XML for documentation purposes.

Common Programming Error 18.1

The placement of any characters, including whitespace, before the XML declaration is a syntax error.

Portability Tip 18.1

Although the XML declaration is optional, documents should include the declaration to identify the version of XML used. Otherwise, in the future, a document that lacks an XML declaration might be assumed to conform to the latest version of XML, and errors could result.

In XML, data are marked up using *tags*, which are names enclosed in *angle brackets* (**`<>`**). Tags are used in pairs to delimit character data (e.g., **`Simple XML`** in line 8). A tag that begins *markup* (i.e., XML data) is called a *start tag*, whereas a tag that terminates markup is called an *end tag*. Examples of start tags are **`<article>`** and **`<title>`** (lines 6 and 8, respectively). End tags differ from start tags in that they contain a *forward slash* (**`/`**) character immediately after the **`<`** character. Examples of end tags are **`</title>`** and **`</article>`** (lines 8 and 23, respectively). XML documents can contain any number of tags.

Common Programming Error 18.2

Failure to provide a corresponding end tag for a start tag is a syntax error.

Individual units of markup (i.e., everything included between a start tag and its corresponding end tag) are called *elements*. An XML document includes one element (called a *root element*) that contains every other element. The root element must be the first element after the XML declaration. In Fig. 18.1, **`article`** (line 6) is the root element. Elements are *nested* within each other to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create explicit relationships between data. For example, elements `title`, `date`, `author`, `summary` and `content` are nested within `article`. Elements `firstName` and `lastName` are nested within `author`.

Common Programming Error 18.3

Attempting to create more than one root element in an XML document is a syntax error.

Element `title` (line 8) contains the title of the article, `Simple XML`, as character data. Similarly, `date` (line 10), `summary` (line 17) and `content` (lines 19–21) contain as character data the date, summary and content, respectively. XML element names can be of any length and may contain letters, digits, underscores, hyphens and periods—they must begin with a letter or an underscore.

Common Programming Error 18.4

XML is case sensitive. The use of the wrong case for an XML element name is a syntax error.

By itself, this document is simply a text file named `article.xml`. Although it is not required, most XML documents end in the file extension *`.xml`*. The processing of XML documents requires a program called an *XML parser* also called *XML processors*. Parsers are responsible for checking an XML document's syntax and making the XML document's data available to applications. Often, XML parsers are built into applications such as Visual Studio or available for download over the Internet. Popular parsers include Microsoft's *msxml*, the Apache Software Foundation's *Xerces* and IBM's *XML4J*. In this chapter, we use msxml.

When the user loads `article.xml` into Internet Explorer (IE)[1], msxml parses the document and passes the parsed data to IE. IE then uses a built-in *style sheet* to format the data. Notice that the resulting format of the data (Fig. 18.2) is similar to the format of the XML document shown in Fig. 18.1. As we soon demonstrate, style sheets play an important and powerful role in the transformation of XML data into formats suitable for display.

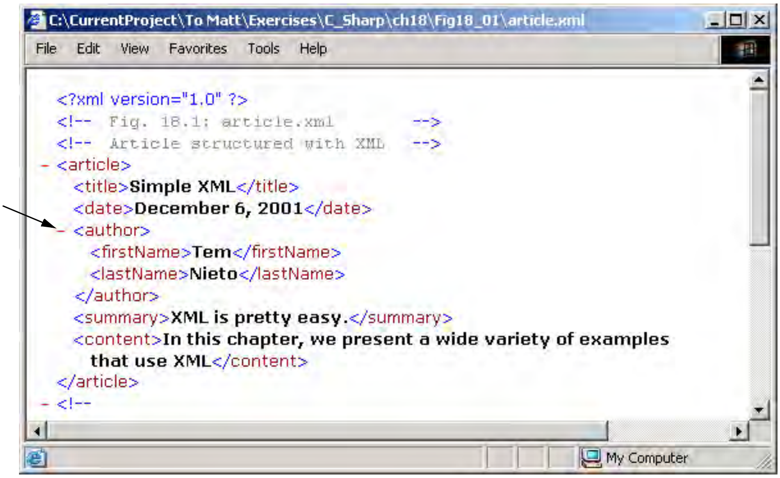
Notice the minus (**–**) and plus (**+**) signs in Fig. 18.2. Although these are not part of the XML document, IE places them next to all *container elements* (i.e., elements that contain other elements). Container elements also are called *parent elements*. A minus sign indicates that the parent element's *child elements* (i.e., nested elements) are being displayed. When clicked, a minus sign becomes a plus sign (which collapses the container element and hides all children). Conversely, clicking a plus sign expands the container element and changes the plus sign to a minus sign. This behavior is similar to the viewing of the directory structure on a Windows system using Windows Explorer. In fact, a directory structure often is modeled as a series of tree structures, in which each drive letter (e.g., `C:`, etc.) represents the *root* of a tree. Each folder is a *node* in the tree. Parsers often place XML data into trees to facilitate efficient manipulation, as discussed in Section 18.4.

Common Programming Error 18.5

Nesting XML tags improperly is a syntax error. For example, `<x><y>hello</x></y>` is a error, because the `</y>` tag must precede the `</x>` tag.

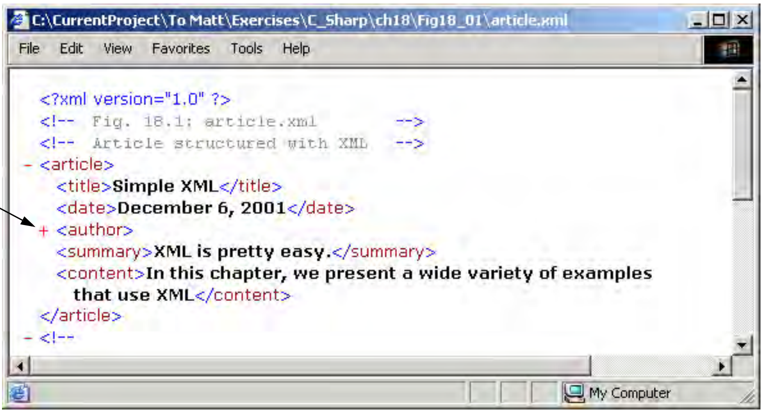
---

[1] IE 5 and higher.


C:\CurrentProject\To Matt\Exercises\C_Sharp\ch18\Fig18_01\article.xml
File
Edit
View
Favorites
Tools
Help
My Computer

Minus sign

```
  <?xml version="1.0" ?>
  <!--  Fig. 18.1: article.xml        -->
  <!--  Article structured with XML   -->
- <article>
    <title>Simple XML</title>
    <date>December 6, 2001</date>
  - <author>
      <firstName>Tem</firstName>
      <lastName>Nieto</lastName>
    </author>
    <summary>XML is pretty easy.</summary>
    <content>In this chapter, we present a wide variety of examples
      that use XML</content>
  </article>
- <!--
```


C:\CurrentProject\To Matt\Exercises\C_Sharp\ch18\Fig18_01\article.xml
File
Edit
View
Favorites
Tools
Help
My Computer

Plus sign

```
  <?xml version="1.0" ?>
  <!--  Fig. 18.1: article.xml        -->
  <!--  Article structured with XML   -->
- <article>
    <title>Simple XML</title>
    <date>December 6, 2001</date>
  + <author>
    <summary>XML is pretty easy.</summary>
    <content>In this chapter, we present a wide variety of examples
      that use XML</content>
  </article>
- <!--
```

Fig. 18.2 `article.xml` displayed by Internet Explorer.

We now present a second XML document (Fig. 18.3), which marks up a business letter. This document contains significantly more data than did the previous XML document.

```
<?xml version = "1.0"?>

<!-- Fig. 18.3: letter.xml              -->
<!-- Business letter formatted with XML -->

<letter>
   <contact type = "from">
      <name>Jane Doe</name>
      <address1>Box 12345</address1>
      <address2>15 Any Ave.</address2>
      <city>Othertown</city>
      <state>Otherstate</state>
```

Fig. 18.3 XML to mark up a business letter. (Part 1 of 2.)

```
13        <zip>67890</zip>
14        <phone>555-4321</phone>
15        <flag gender = "F" />
16     </contact>
17
18     <contact type = "to">
19        <name>John Doe</name>
20        <address1>123 Main St.</address1>
21        <address2></address2>
22        <city>Anytown</city>
23        <state>Anystate</state>
24        <zip>12345</zip>
25        <phone>555-1234</phone>
26        <flag gender = "M" />
27     </contact>
28
29     <salutation>Dear Sir:</salutation>
30
31        <paragraph>It is our privilege to inform you about our new
32        database managed with <technology>XML</technology>. This
33        new system allows you to reduce the load on
34        your inventory list server by having the client machine
35        perform the work of sorting and filtering the data.
36        </paragraph>
37
38        <paragraph>Please visit our Web site for availability
39        and pricing.
40        </paragraph>
41
42     <closing>Sincerely</closing>
43
44     <signature>Ms. Doe</signature>
45  </letter>
```

Fig. 18.3 XML to mark up a business letter. (Part 2 of 2.)

Root element **`letter`** (lines 6–45) contains the child elements **`contact`** (lines 7–16 and 18–27), **`salutation`**, **`paragraph`** (lines 31–36 and 38–40), **`closing`** and **`signature`**. In addition to being placed between tags, data also can be placed in *attributes*, which are name-value pairs in start tags. Elements can have any number of attributes in their start tags. The first **`contact`** element (lines 7–16) has attribute **`type`** with attribute *value* **`"from"`**, which indicates that this **`contact`** element marks up information about the letter's sender. The second **`contact`** element (lines 18–27) has attribute **`type`** with value **`"to"`**, which indicates that this **`contact`** element marks up information about the letter's recipient. Like element names, attribute names are case sensitive, can be any length; may contain letters, digits, underscores, hyphens and periods; and must begin with either a letter or underscore character. A **`contact`** element stores a contact's name, address and phone number. Element **`salutation`** (line 29) marks up the letter's salutation. Lines 31–40 mark up the letter's body with **`paragraph`** elements. Elements **`closing`** (line 42) and **`signature`** (line 44) mark up the closing sentence and the signature of the letter's author, respectively.

Common Programming Error 18.6

Failure to enclose attribute values in either double (`""`) or single (`''`) quotes is a syntax error.

Common Programming Error 18.7

Attempting to provide two attributes with the same name for an element is a syntax error.

In line 15, we introduce *empty element* `flag`, which indicates the gender of the contact. Empty elements do not contain character data (i.e., they do not contain text between the start and end tags). Such elements are closed either by placing a slash at the end of the element (as shown in line 15) or by explicitly writing a closing tag, as in

```
<flag gender = "F"></flag>
```

18.3 XML Namespaces

Object-oriented programming languages, such as C# and Visual Basic .NET, provide massive class libraries that group their features into namespaces. These namespaces prevent *naming collisions* between programmer-defined identifiers and identifiers in class libraries. For example, we might use class `Book` to represent information on one of our publications; however, a stamp collector might use class `Book` to represent a book of stamps. A naming collision would occur if we use these two classes in the same assembly, without using namespaces to differentiate them.

Like C#, XML also provides *namespaces*, which provide a means of uniquely identifying XML elements. In addition, XML-based languages—called *vocabularies*, such as XML Schema (Section 18.5), Extensible Stylesheet Language (Section 18.6) and BizTalk (Section 18.7)—often use namespaces to identify their elements.

Elements are differentiated via *namespace prefixes*, which identify the namespace to which an element belongs. For example,

```
<deitel:book>C# How to Program</deitel:book>
```

qualifies element `book` with namespace prefix `deitel`. This indicates that element `book` is part of namespace `deitel`. Document authors can use any name for a namespace prefix except the reserved namespace prefix *`xml`*.

Common Programming Error 18.8

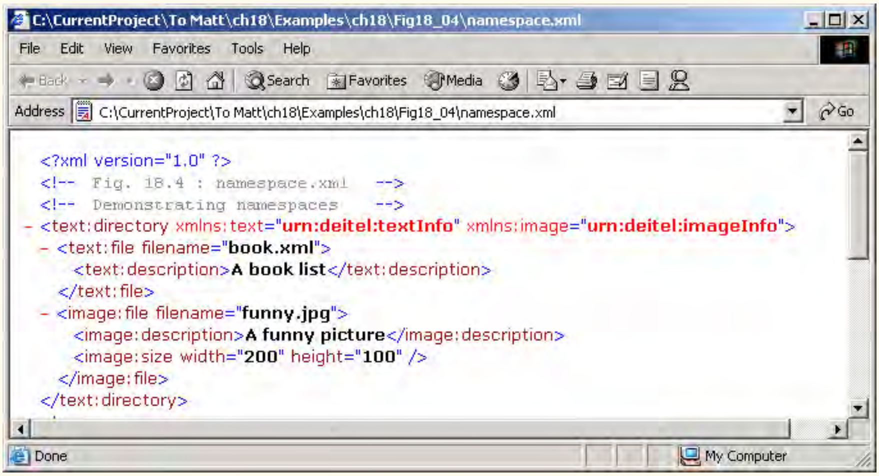
Attempting to create a namespace prefix named `xml` in any mixture of case is a syntax error.

The mark up in Fig. 18.4 demonstrates the use of namespaces. This XML document contains two `file` elements that are differentiated using namespaces.

```
<?xml version = "1.0"?>

<!-- Fig. 18.4: namespace.xml -->
<!-- Demonstrating namespaces -->
```

Fig. 18.4 XML namespaces demonstration. (Part 1 of 2.)

```
 5
 6  <text:directory xmlns:text = "urn:deitel:textInfo"
 7     xmlns:image = "urn:deitel:imageInfo">
 8
 9     <text:file filename = "book.xml">
10        <text:description>A book list</text:description>
11     </text:file>
12
13     <image:file filename = "funny.jpg">
14        <image:description>A funny picture</image:description>
15        <image:size width = "200" height = "100" />
16     </image:file>
17
18  </text:directory>
```


C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_04\namespace.xml
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_04\namespace.xml
Go
Done
My Computer

```
  <?xml version="1.0" ?>
  <!-- Fig. 18.4 : namespace.xml   -->
  <!-- Demonstrating namespaces    -->
- <text:directory xmlns:text="urn:deitel:textInfo" xmlns:image="urn:deitel:imageInfo">
  - <text:file filename="book.xml">
      <text:description>A book list</text:description>
    </text:file>
  - <image:file filename="funny.jpg">
      <image:description>A funny picture</image:description>
      <image:size width="200" height="100" />
    </image:file>
  </text:directory>
```

Fig. 18.4 XML namespaces demonstration. (Part 2 of 2.)

Software Engineering Observation 18.1

A programmer has the option of qualifying an attribute with a namespace prefix. However, it is not required, because attributes always are associated with elements.

Lines 6–7 use attribute ***`xmlns`*** to create two namespace prefixes: **`text`** and **`image`**. Each namespace prefix is bound to a series of characters called a *uniform resource identifier (URI)* that uniquely identifies the namespace. Document authors create their own namespace prefixes and URIs.

To ensure that namespaces are unique, document authors must provide unique URIs. Here, we use the text **`urn:deitel:textInfo`** and **`urn:deitel:imageInfo`** as URIs. A common practice is to use *Universal Resource Locators (URLs)* for URIs, because the domain names (such as, **`www.deitel.com`**) used in URLs are guaranteed to be unique. For example, lines 6–7 could have been written as

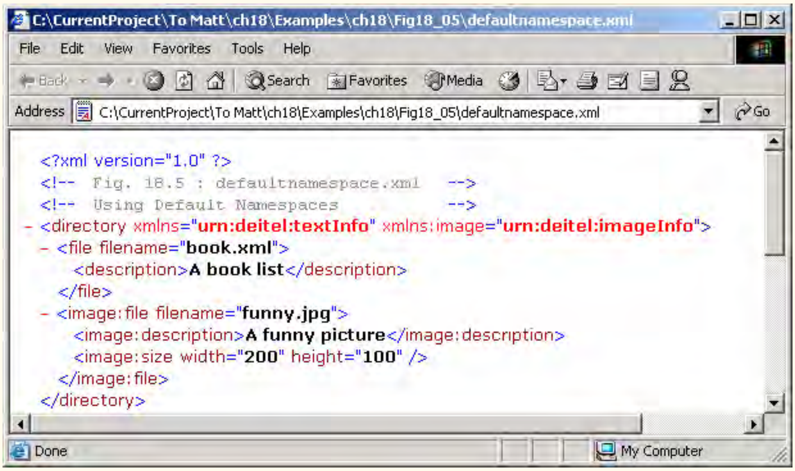
```
<text:directory xmlns:text =
   "http://www.deitel.com/xmlns-text"
   xmlns:image = "http://www.deitel.com/xmlns-image">
```

In this example, we use URLs related to the Deitel & Associates, Inc, domain name to identify namespaces. The parser never visits these URLs—they simply represent a series of characters used to differentiate names. The URLs need not refer to actual Web pages or be formed properly.

Lines 9–11 use the namespace prefix **`text`** to qualify elements **`file`** and **`description`** as belonging to the namespace **`"urn:deitel:textInfo"`**. Notice that the namespace prefix **`text`** is applied to the end tags as well. Lines 13–16 apply namespace prefix **`image`** to elements **`file`**, **`description`** and **`size`**.

To eliminate the need to precede each element with a namespace prefix, document authors can specify a *default namespace*. Figure 18.5 demonstrates the creation and use of default namespaces.

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 18.5: defaultnamespace.xml -->
 4  <!-- Using default namespaces         -->
 5
 6  <directory xmlns = "urn:deitel:textInfo"
 7     xmlns:image = "urn:deitel:imageInfo">
 8
 9     <file filename = "book.xml">
10        <description>A book list</description>
11     </file>
12
13     <image:file filename = "funny.jpg">
14        <image:description>A funny picture</image:description>
15        <image:size width = "200" height = "100" />
16     </image:file>
17
18  </directory>
```


C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_05\defaultnamespace.xml
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_05\defaultnamespace.xml Go
Done
My Computer

```
  <?xml version="1.0" ?>
  <!--  Fig. 18.5 : defaultnamespace.xml   -->
  <!--  Using Default Namespaces           -->
- <directory xmlns="urn:deitel:textInfo" xmlns:image="urn:deitel:imageInfo">
  - <file filename="book.xml">
      <description>A book list</description>
    </file>
  - <image:file filename="funny.jpg">
      <image:description>A funny picture</image:description>
      <image:size width="200" height="100" />
    </image:file>
  </directory>
```

Fig. 18.5 Default namespaces demonstration.

Line 6 declares a default namespace using attribute **`xmlns`** with a URI as its value. Once we define this default namespace, child elements belonging to the namespace need not be qualified by a namespace prefix. Element **`file`** (line 9–11) is in the namespace **`urn:deitel:textInfo`**. Compare this to Fig. 18.4, where we prefixed **`file`** and **`description`** with **`text`** (lines 9–11).

The default namespace applies to the **`directory`** element and all elements that are not qualified with a namespace prefix. However, we can use a namespace prefix to specify a different namespace for particular elements. For example, the **`file`** element in line 13 is prefixed with **`image`** to indicate that it is in the namespace **`urn:deitel:imageInfo`**, rather than the default namespace.

18.4 Document Object Model (DOM)

Although XML documents are text files, retrieving data from them via sequential-file access techniques is neither practical nor efficient, especially in situations where data must be added or deleted dynamically.

Upon successful parsing of documents, some XML parsers store document data as tree structures in memory. Figure 18.6 illustrates the tree structure for the document **`article.xml`** discussed in Fig. 18.1. This hierarchical tree structure is called a *Document Object Model (DOM)* tree, and an XML parser that creates this type of structure is known as a *DOM parser*. The DOM tree represents each component of the XML document (e.g., **`article`**, **`date`**, **`firstName`**, etc.) as a node in the tree. Nodes (such as, **`author`**) that contain other nodes (called *child nodes*) are called *parent nodes*. Nodes that have the same parent (such as, **`firstName`** and **`lastName`**) are called *sibling nodes*. A node's *descendant nodes* include that node's children, its children's children and so on. Similarly, a node's *ancestor nodes* include that node's parent, its parent's parent and so on. Every DOM tree has a single *root node* that contains all other nodes in the document, such as comments, elements, etc.

Classes for creating, reading and manipulating XML documents are located in the C# namespace ***`System.Xml`***. This namespace also contains additional namespaces that contain other XML-related operations.

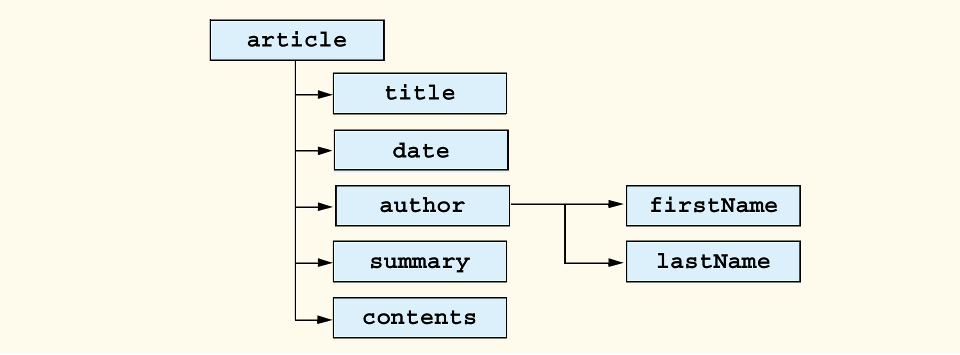


Fig. 18.6 Tree structure for Fig. 18.1.

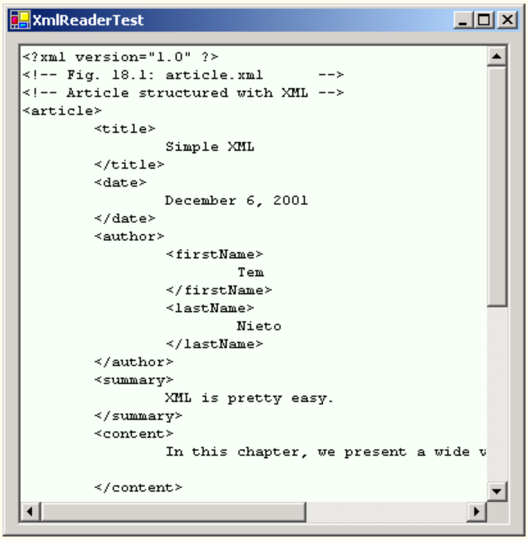
In this section, we present several examples that use DOM trees. Our first example, the program in Fig. 18.7, loads the XML document presented in Fig. 18.1 and displays its data in a text box. This example uses class ***`XmlNodeReader`*** which is derived from ***`XmlReader`***, which iterates through each node in the XML document. Class **`XmlReader`** is an **`abstract`** class that defines the interface for reading XML documents.

```
// Fig. 18.7: XmlReaderTest.cs
// Reading an XML document.

using System;
using System.Windows.Forms;
using System.Xml;

public class XmlReaderTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox outputTextBox;
   private System.ComponentModel.Container components = null;

   public XmlReaderTest()
   {
      InitializeComponent();

      // reference to "XML document"
      XmlDocument document = new XmlDocument();
      document.Load( "..\\..\\article.xml" );

      // create XmlNodeReader for document
      XmlNodeReader reader = new XmlNodeReader( document );

      // show form before outputTextBox is populated
      this.Show();

      // tree depth is -1, no indentation
      int depth = -1;

      // display each node's content
      while ( reader.Read() )
      {
         switch ( reader.NodeType )
         {
            // if Element, display its name
            case XmlNodeType.Element:

               // increase tab depth
               depth++;
               TabOutput( depth );
               outputTextBox.Text += "<" + reader.Name + ">" +
                  "\r\n";

```

Fig. 18.7 `XmlNodeReader` used to iterate through an XML document. (Part 1 of 3.)

```
               // if empty element, decrease depth
               if ( reader.IsEmptyElement )
                  depth--;

               break;

            // if Comment, display it
            case XmlNodeType.Comment:
               TabOutput( depth );
               outputTextBox.Text +=
                  "<!--" + reader.Value + "-->\r\n";
               break;

            // if Text, display it
            case XmlNodeType.Text:
               TabOutput( depth );
               outputTextBox.Text += "\t" + reader.Value +
                  "\r\n";
               break;

            // if XML declaration, display it
            case XmlNodeType.XmlDeclaration:
               TabOutput( depth );
               outputTextBox.Text += "<?" + reader.Name + " "
                  + reader.Value + " ?>\r\n";
               break;

            // if EndElement, display it and decrement depth
            case XmlNodeType.EndElement:
               TabOutput( depth );
               outputTextBox.Text += "</" + reader.Name
                  + ">\r\n";
               depth--;
               break;
         } // end switch statement
      } // end while loop
   } // End XmlReaderTest constructor

   // insert tabs
   private void TabOutput( int number )
   {
      for ( int i = 0; i < number; i++ )
         outputTextBox.Text += "\t";
   } // end TabOutput

   // Windows Form Designer generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new XmlReaderTest() );
   } // end Main
} // end XmlReaderTest
```

Fig. 18.7 **`XmlNodeReader`** used to iterate through an XML document. (Part 2 of 3.)



```
<?xml version="1.0" ?>
<!-- Fig. 18.1: article.xml      -->
<!-- Article structured with XML -->
<article>
        <title>
                Simple XML
        </title>
        <date>
                December 6, 2001
        </date>
        <author>
                <firstName>
                        Tem
                </firstName>
                <lastName>
                        Nieto
                </lastName>
        </author>
        <summary>
                XML is pretty easy.
        </summary>
        <content>
                In this chapter, we present a wide v

        </content>
```

Fig. 18.7 `XmlNodeReader` used to iterate through an XML document. (Part 3 of 3.)

Line 6 includes the **`System.Xml`** namespace, which contains the XML classes used in this example. Line 18 creates a reference to an ***`XmlDocument`*** object that conceptually represents an empty XML document. The XML document **`article.xml`** is parsed and loaded into this **`XmlDocument`** object when method ***`Load`*** is invoked in line 19. Once an XML document is loaded into an **`XmlDocument`**, its data can be read and manipulated programmatically. In this example, we read each node in the **`XmlDocument`**, which is the DOM tree. In successive examples, we demonstrate how to manipulate node values.

In line 22, we create an **`XmlNodeReader`** and assign it to reference **`reader`**, which enables us to read one node at a time from the **`XmlDocument`**. Method ***`Read`*** of **`XmlReader`** reads one node from the DOM tree. Placing this statement in the **`while`** loop (lines 31–78) makes **`reader`** **`Read`** all the document nodes. The **`switch`** statement (lines 33–77) processes each node. Either the ***`Name`*** property (line 41), which contains the node's name, or the ***`Value`*** property (line 53), which contains the node's data, is formatted and concatenated to the **`string`** assigned to the text box **`Text`** property. The **`NodeType`** property contains the node type (specifying whether the node is an element, comment, text, etc.). Notice that each **`case`** specifies a node type, using ***`XmlNodeType`*** enumeration constants.

Notice that the displayed output emphasizes the structure of the XML document. Variable **`depth`** (line 28) sets the number of tab characters used to indent each element. The depth is incremented each time an **`Element`** type is encountered and is decremented each time an **`EndElement`** or empty element is encountered. We use a similar technique in the next example to emphasize the tree structure of the XML document in the display.

Notice that our line breaks use the character sequence **`"\r\n"`**, which denotes a carriage return followed by a line feed. This is the standard line break for Windows-based applications and controls.

The C# program in Fig. 18.8 demonstrates how to manipulate DOM trees programmatically. This program loads **`letter.xml`** (Fig. 18.3) into the DOM tree and then creates a second DOM tree that duplicates the DOM tree containing `letter.xml`'s contents. The GUI for this application contains a text box, a `TreeView` control and three buttons—**Build**, **Print** and **Reset**. When clicked, **Build** copies `letter.xml` and displays the document's tree structure in the `TreeView` control, **Print** displays the XML element values and names in a text box and **Reset** clears the `TreeView` control and text box content.

Lines 20 and 23 create references to `XmlDocument`s `source` and `copy`. Line 32 assigns a new `XmlDocument` object to reference `source`. Line 33 then invokes method `Load` to parse and load `letter.xml`. We discuss reference `copy` shortly.

Unfortunately, `XmlDocument`s do not provide any features for displaying their content graphically. In this example, we display the document's contents via a `TreeView` control. We use objects of class ***`TreeNode`*** to represent each node in the tree. Class `TreeView` and class `TreeNode` are part of the `System.Windows.Forms` namespace. `TreeNode`s are added to the `TreeView` to emphasize the structure of the XML document.

```
 1  // Fig. 18.8: XmlDom.cs
 2  // Demonstrates DOM tree manipulation.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml;
 7  using System.IO;
 8  using System.CodeDom.Compiler;  // contains TempFileCollection
 9
10  // Class XmlDom demonstrates the DOM
11  public class XmlDom : System.Windows.Forms.Form
12  {
13     private System.Windows.Forms.Button buildButton;
14     private System.Windows.Forms.Button printButton;
15     private System.Windows.Forms.TreeView xmlTreeView;
16     private System.Windows.Forms.TextBox consoleTextBox;
17     private System.Windows.Forms.Button resetButton;
18     private System.ComponentModel.Container components = null;
19
20     private XmlDocument source; // reference to "XML document"
21
22     // reference copy of source's "XML document"
23     private XmlDocument copy;
24
25     private TreeNode tree; // TreeNode reference
26
27     public XmlDom()
28     {
29        InitializeComponent();
30
31        // create XmlDocument and load letter.xml
32        source = new XmlDocument();
33        source.Load( "..\\..\\letter.xml" );
34
35        // initialize references to null
36        copy = null;
```

Fig. 18.8 DOM structure of an XML document illustrated by a class. (Part 1 of 6.)

```
      tree = null;
   } // end XmlDom

   [STAThread]
   static void Main()
   {
      Application.Run( new XmlDom() );
   }

   // event handler for buildButton click event
   private void buildButton_Click( object sender,
      System.EventArgs e )
   {
      // determine if copy has been built already
      if ( copy != null )
         return;  // document already exists

      // instantiate XmlDocument and TreeNode
      copy = new XmlDocument();
      tree = new TreeNode();

      // add root node name to TreeNode and add
      // TreeNode to TreeView control
      tree.Text = source.Name;      // assigns #root
      xmlTreeView.Nodes.Add( tree );

      // build node and tree hierarchy
      BuildTree( source, copy, tree );

      printButton.Enabled = true;
      resetButton.Enabled = true;
   } // end buildButton_Click

   // event handler for printButton click event
   private void printButton_Click( object sender,
      System.EventArgs e )
   {
      // exit if copy does not reference an XmlDocument
      if ( copy == null )
         return;

      // create temporary XML file
      TempFileCollection file = new TempFileCollection();

      // create file that is deleted at program termination
      file.AddExtension( "xml", false );
      string[] filename = new string[ 1 ];
      file.CopyTo( filename, 0 );

      // write XML data to disk
      XmlTextWriter writer = new XmlTextWriter( filename[ 0 ],
         System.Text.Encoding.UTF8 );
      copy.WriteTo( writer );
```

Fig. 18.8 DOM structure of an XML document illustrated by a class. (Part 2 of 6.)

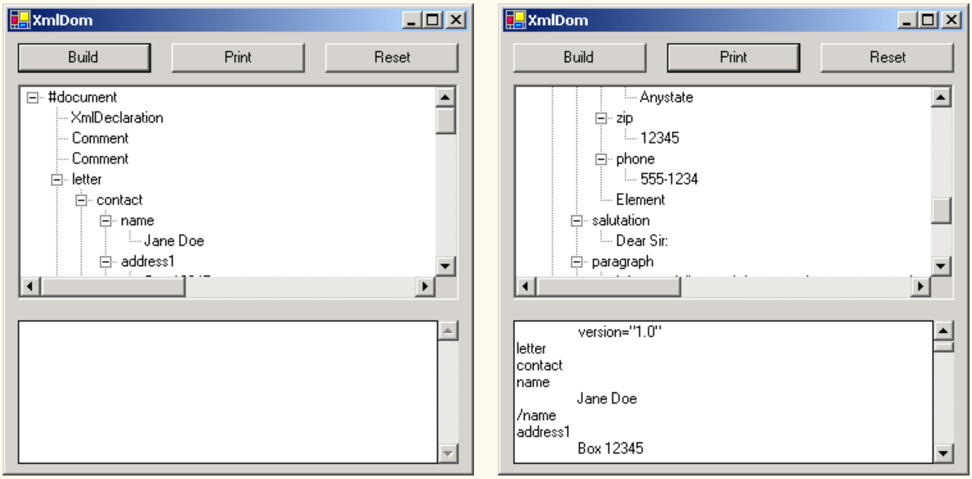
```
      writer.Close();

      // parse and load temporary XML document
      XmlTextReader reader = new XmlTextReader( filename[ 0 ] );

      // read, format and display data
      while( reader.Read() )
      {
         if ( reader.NodeType == XmlNodeType.EndElement )
            consoleTextBox.Text += "/";

         if ( reader.Name != String.Empty )
            consoleTextBox.Text += reader.Name + "\r\n";

         if ( reader.Value != String.Empty )
            consoleTextBox.Text += "\t" + reader.Value +
               "\r\n";
      } // end while

      reader.Close();
   } // end printButton_Click

   // handle resetButton click event
   private void resetButton_Click( object sender,
      System.EventArgs e )
   {
      // remove TreeView nodes
      if ( tree != null )
         xmlTreeView.Nodes.Remove( tree );

      xmlTreeView.Refresh(); // force TreeView update

      // delete XmlDocument and tree
      copy = null;
      tree = null;

      consoleTextBox.Text = "";  // clear text box

      printButton.Enabled = false;
      resetButton.Enabled = false;

   } // end resetButton_Click

   // construct DOM tree
   private void BuildTree( XmlNode xmlSourceNode,
      XmlNode document, TreeNode treeNode )
   {
      // create XmlNodeReader to access XML document
      XmlNodeReader nodeReader = new XmlNodeReader(
         xmlSourceNode );

      // represents current node in DOM tree
      XmlNode currentNode = null;
```

Fig. 18.8 DOM structure of an XML document illustrated by a class. (Part 3 of 6.)

```

         // treeNode to add to existing tree
         TreeNode newNode = new TreeNode();

         // references modified node type for CreateNode
         XmlNodeType modifiedNodeType;

         while ( nodeReader.Read() )
         {
            // get current node type
            modifiedNodeType = nodeReader.NodeType;

            // check for EndElement, store as Element
            if ( modifiedNodeType == XmlNodeType.EndElement )
               modifiedNodeType = XmlNodeType.Element;

            // create node copy
            currentNode = copy.CreateNode( modifiedNodeType,
               nodeReader.Name, nodeReader.NamespaceURI );

            // build tree based on node type
            switch ( nodeReader.NodeType )
            {
               // if Text node, add its value to tree
               case XmlNodeType.Text:
                  newNode.Text = nodeReader.Value;
                  treeNode.Nodes.Add( newNode );

                  // append Text node value to currentNode data
                  ( ( XmlText ) currentNode ).AppendData(
                     nodeReader.Value );
                  document.AppendChild( currentNode );
                  break;

               // if EndElement, move up tree
               case XmlNodeType.EndElement:
                  document = document.ParentNode;
                  treeNode = treeNode.Parent;
                  break;

               // if new element, add name and traverse tree
               case XmlNodeType.Element:

                  // determine if element contains content
                  if ( !nodeReader.IsEmptyElement )
                  {
                     // assign node text, add newNode as child
                     newNode.Text = nodeReader.Name;
                     treeNode.Nodes.Add( newNode );

                     // set treeNode to last child
                     treeNode = newNode;

```

Fig. 18.8 DOM structure of an XML document illustrated by a class. (Part 4 of 6.)

```
                  document.AppendChild( currentNode );
                  document = document.LastChild;
               }
               else // do not traverse empty elements
               {
                  // assign NodeType string to newNode
                  newNode.Text =
                     nodeReader.NodeType.ToString();

                  treeNode.Nodes.Add( newNode );
                  document.AppendChild( currentNode );
               }

               break;

            // all other types, display node type
            default:
               newNode.Text = nodeReader.NodeType.ToString();
               treeNode.Nodes.Add( newNode );
               document.AppendChild( currentNode );
               break;
         }  // end switch

         newNode = new TreeNode();
      } // end while

      // update the TreeView control
      xmlTreeView.ExpandAll();
      xmlTreeView.Refresh();

   } // end BuildTree
} // end XmlDom
```

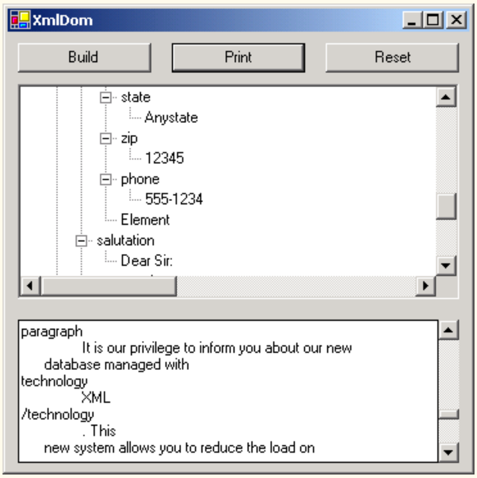


```
#document
   XmlDeclaration
   Comment
   Comment
   letter
      contact
         name
            Jane Doe
         address1
```

```
            Anystate
         zip
            12345
         phone
            555-1234
         Element
      salutation
         Dear Sir:
      paragraph
```

```
          version="1.0"
letter
contact
name
          Jane Doe
/name
address1
          Box 12345
```

Fig. 18.8 DOM structure of an XML document illustrated by a class. (Part 5 of 6.)



```
state
    Anystate
zip
    12345
phone
    555-1234
Element
salutation
    Dear Sir:
```

```
paragraph
        It is our privilege to inform you about our new
    database managed with
technology
        XML
/technology
        . This
    new system allows you to reduce the load on
```

Fig. 18.8 DOM structure of an XML document illustrated by a class. (Part 6 of 6.)

When clicked, button **Build** triggers event handler **`buildButton_Click`** (lines 47–68), which copies **`letter.xml`** dynamically. The new **`XmlDocument`** and **`TreeNode`**s (i.e., the nodes used for graphical representation in the **`TreeView`**) are created in lines 55–56. Line 60 retrieves the **`Name`** of the node referenced by **`source`** (i.e., **`#root`**, which represents the document root) and assigns it to **`tree`**'s **`Text`** property. This **`TreeNode`** then is inserted into the **`TreeView`** control's node list. Method **`Add`** is called to add each new **`TreeNode`** to the **`TreeView`**'s **`Nodes`** collection. Line 64 calls method **`BuildTree`** to copy the **`XMLDocument`** referenced by **`source`** and to update the **`TreeView`**.

Method **`BuildTree`** (line 134–226) receives an **`XmlNode`** representing the source node, an empty **`XmlNode`** and a **`treeNode`** to place in the DOM tree. Parameter **`treeNode`** references the current location in the tree (i.e., the **`TreeNode`** most recently added to the **`TreeView`** control). Lines 138–139 instantiate a new **`XmlNodeReader`** for iterating through the DOM tree. Lines 142–145 declare **`XmlNode`** and **`TreeNode`** references that indicate the next nodes added to **`document`** (i.e., the DOM tree referenced by **`copy`**) and **`treeNode`**. Lines 150–220 iterate through each node in the tree.

Lines 153–161 create a node containing a copy of the current **`nodeReader`** node. Method ***`CreateNode`*** of **`XmlDocument`** takes a **`NodeType`**, a **`Name`** and a ***`NamespaceURI`*** as arguments. The **`NodeType`** cannot be an **`EndElement`**. If the **`NodeType`** is of an **`EndElement`** type, lines 156–157 assign **`modifiedNodeType`** type **`Element`**.

The **`switch`** statement in lines 164–217 determines the node type, creates and adds nodes to the **`TreeView`** and updates the DOM tree.When a text node is encountered, the new **`TreeNode`**'s **`newNode`**'s **`Text`** property is assigned the current node's value. This **`TreeNode`** is added to the **`TreeView`** control. In lines 172–174, we downcast **`currentNode`** to **`XmlText`** and append the node's value. The **`currentNode`** then is appended to the **`document`**. Lines 171–174 match an **`EndElement`** node type. This **`case`** moves up the tree, because the end of an element has been encountered. The ***`ParentNode`*** and ***`Parent`*** properties retrieve the **`document`**'s and **`treeNode`**'s parents, respectively.

Line 177 matches **`Element`** node types. Each nonempty **`Element NodeType`** (line 180) increases the depth of the tree; thus, we assign the current **`nodeReader Name`** to the **`newNode`**'s **`Text`** property and add the **`newNode`** to the **`treeNode`** node list. Lines 187–190 reorder the nodes in the node list to ensure that **`newNode`** is the last **`TreeNode`** in the node list. **`XmlNode currentNode`** is appended to **`document`** as the last child, and **`document`** is set to its ***`LastChild`***, which is the child we just added. If it is an empty element (line 192), we assign to the **`newNode`**'s **`Text`** property the **`string`** representation of the **`NodeType`**. Next, the **`newNode`** is added to the **`treeNode`** node list. Line 199 appends the **`currentNode`** to the **`document`**. The **`default`** case assigns the string representation of the node type to the **`NewNode Text`** property, adds the **`newNode`** to the **`TreeNode`** node list and appends the **`currentNode`** to the **`document`**.

After building the DOM trees, the **`TreeNode`** node list displays in the **`TreeView`** control. Clicking the nodes (i.e., the **+** or **-** boxes) in the **`TreeView`** either expands or collapses them. Clicking **Print** invokes event handler **`printButton_Click`** (line 71). Lines 79–84 create a temporary file for storing the XML. Line 87 creates an **`XmlTextWriter`** for streaming the XML data to disk. Method ***`WriteTo`*** is called to write the XML representation to the ***`XmlTextWriter`*** stream (line 89). Line 93 creates an ***`XmlTextReader`*** to read from the file. The **`while`** loop (line 96–107) reads each node in the DOM tree and writes tag names and character data to the text box. If it is an end element, a slash is concatenated. If the node has a **`Name`** or **`Value`**, that name or value is concatenated to the textbox text.

The **Reset** button's event handler, **`resetButton_Click`**, deletes both dynamically generated trees and updates the **`TreeView`** control's display. Reference **`copy`** is assigned **`null`** (to allow its tree to be garbage collected in line 123), and the **`TreeNode`** node list reference **`tree`** is assigned **`null`**.

Although **`XmlReader`** includes methods for reading and modifying node values, it is not the most efficient means of locating data in a DOM tree. The .NET framework provides class ***`XPathNavigator`*** in the ***`System.Xml.XPath`*** namespace for iterating through node lists that match search criteria, which are written as an *XPath expression*. XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies (such as, XSLT, discussed in Section 18.6).

Figure 18.9 demonstrates how to navigate through an XML document with an **`XPathNavigator`**. Like Fig. 18.8, this program uses a **`TreeView`** control and **`TreeNode`** objects to display the XML document's structure. However, instead of displaying the entire DOM tree, the **`TreeNode`** node list is updated each time the **`XPathNavigator`** is positioned to a new node. Nodes are added to and deleted from the **`TreeView`** to reflect the **`XPathNavigator`**'s location in the DOM tree. The XML document **`sports.xml`** that we use in this example is presented in Figure 18.10.

This program loads XML document **`sports.xml`** into an ***`XPathDocument`*** object by passing the document's file name to the **`XPathDocument`** constructor (line 36). Method ***`CreateNavigator`*** (line 39) creates and returns an **`XPathNavigator`** reference to the **`XPathDocument`**'s tree structure.

The navigation methods of **`XPathNavigator`** used in Fig. 18.9 are ***`MoveToFirstChild`*** (line 66), ***`MoveToParent`*** (line 94), ***`MoveToNext`*** (line 122) and ***`MoveToPrevious`*** (line 151). Each method performs the action that its name implies. Method **`MoveToFirstChild`** moves to the first child of the node referenced by the

`XPathNavigator`, **`MoveToParent`** moves to the parent node of the node referenced by the **`XPathNavigator`**, **`MoveToNext`** moves to the next sibling of the node referenced by the **`XPathNavigator`** and **`MoveToPrevious`** moves to the previous sibling of the node referenced by the **`XPathNavigator`**. Each method returns a **`bool`** indicating whether the move was successful. In this example, we display a warning in a **`MessageBox`** whenever a move operation fails. Furthermore, each of these methods is called in the event handler of the button that matches its name (e.g., button **First Child** triggers **`firstChildButton_Click`**, which calls **`MoveToFirstChild`**).

Whenever we move forward via the **`XPathNavigator`**, as with **`MoveToFirstChild`** and **`MoveToNext`**, nodes are added to the **`TreeNode`** node list. Method **`DetermineType`** is a **`private`** method (defined in lines 208–229) that determines whether to assign the **`Node`**'s ***`Name`*** property or ***`Value`*** property to the **`TreeNode`** (lines 218 and 225). Whenever **`MoveToParent`** is called, all children of the parent node are removed from the display. Similarly, a call to **`MoveToPrevious`** removes the current sibling node. Note that the nodes are removed only from the **`TreeView`**, not from the tree representation of the document.

The other event handler corresponds to button **Select** (line 173–174). Method **`Select`** (line 182) takes search criteria in the form of either an ***`XPathExpression`*** or a **`string`** that represents an XPath expression and returns as an **`XPathNodeIterator`** object any nodes that match the search criteria. The XPath expressions provided by this program's combo box are summarized in Fig. 18.11.

Method **`DisplayIterator`** (defined in lines 195–204) appends the node values from the given **`XPathNodeIterator`** to the **`selectTreeViewer`** text box. Note that we call the **`string`** method **`Trim`** to remove unnecessary whitespace. Method ***`MoveNext`*** (line 200) advances to the next node, which can be accessed via property ***`Current`*** (line 202).

```
 1  // Fig. 18.9: PathNavigator.cs
 2  // Demonstrates Class XPathNavigator.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml.XPath; // contains XPathNavigator
 7
 8  public class PathNavigator : System.Windows.Forms.Form
 9  {
10     private System.Windows.Forms.Button firstChildButton;
11     private System.Windows.Forms.Button parentButton;
12     private System.Windows.Forms.Button nextButton;
13     private System.Windows.Forms.Button previousButton;
14     private System.Windows.Forms.Button selectButton;
15     private System.Windows.Forms.TreeView pathTreeViewer;
16     private System.Windows.Forms.ComboBox selectComboBox;
17     private System.ComponentModel.Container components = null;
18     private System.Windows.Forms.TextBox selectTreeViewer;
19     private System.Windows.Forms.GroupBox navigateBox;
20     private System.Windows.Forms.GroupBox locateBox;
21
```

Fig. 18.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 1 of 7.)

```
      // navigator to traverse document
      private XPathNavigator xpath;

      // references document for use by XPathNavigator
      private XPathDocument document;

      // references TreeNode list used by TreeView control
      private TreeNode tree;

      public PathNavigator()
      {
         InitializeComponent();

         // load XML document
         document = new XPathDocument( "..\\..\\sports.xml" );

         // create navigator
         xpath = document.CreateNavigator();

         // create root node for TreeNodes
         tree = new TreeNode();

         tree.Text = xpath.NodeType.ToString(); // #root
         pathTreeViewer.Nodes.Add( tree );      // add tree

         // update TreeView control
         pathTreeViewer.ExpandAll();
         pathTreeViewer.Refresh();
         pathTreeViewer.SelectedNode = tree;    // highlight root
      } // end constructor

      [STAThread]
      static void Main()
      {
         Application.Run( new PathNavigator() );
      }

      // traverse to first child
      private void firstChildButton_Click( object sender,
         System.EventArgs e )
      {
         TreeNode newTreeNode;

         // move to first child
         if ( xpath.MoveToFirstChild() )
         {
            newTreeNode = new TreeNode(); // create new node

            // set node's Text property to either
            // navigator's name or value
            DetermineType( newTreeNode, xpath );

```

Fig. 18.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 2 of 7.)

```
            // add node to TreeNode node list
            tree.Nodes.Add( newTreeNode );
            tree = newTreeNode; // assign tree newTreeNode

            // update TreeView control
            pathTreeViewer.ExpandAll();
            pathTreeViewer.Refresh();
            pathTreeViewer.SelectedNode = tree;
         }
         else // node has no children
            MessageBox.Show( "Current Node has no children.",
               "", MessageBoxButtons.OK,
               MessageBoxIcon.Information );
      }

      // traverse to node's parent on parentButton click event
      private void parentButton_Click( object sender,
         System.EventArgs e )
      {
         // move to parent
         if ( xpath.MoveToParent() )
         {
            tree = tree.Parent;

            // get number of child nodes, not including subtrees
            int count = tree.GetNodeCount( false );

            // remove all children
            tree.Nodes.Clear();

            // update TreeView control
            pathTreeViewer.ExpandAll();
            pathTreeViewer.Refresh();
            pathTreeViewer.SelectedNode = tree;
         }
         else // if node has no parent (root node)
            MessageBox.Show( "Current node has no parent.", "",
               MessageBoxButtons.OK,
               MessageBoxIcon.Information );
      }

      // find next sibling on nextButton click event
      private void nextButton_Click( object sender,
         System.EventArgs e )
      {
         TreeNode newTreeNode = null, newNode = null;

         // move to next sibling
         if ( xpath.MoveToNext() )
         {
            newTreeNode = tree.Parent; // get parent node

            newNode = new TreeNode(); // create new node
```

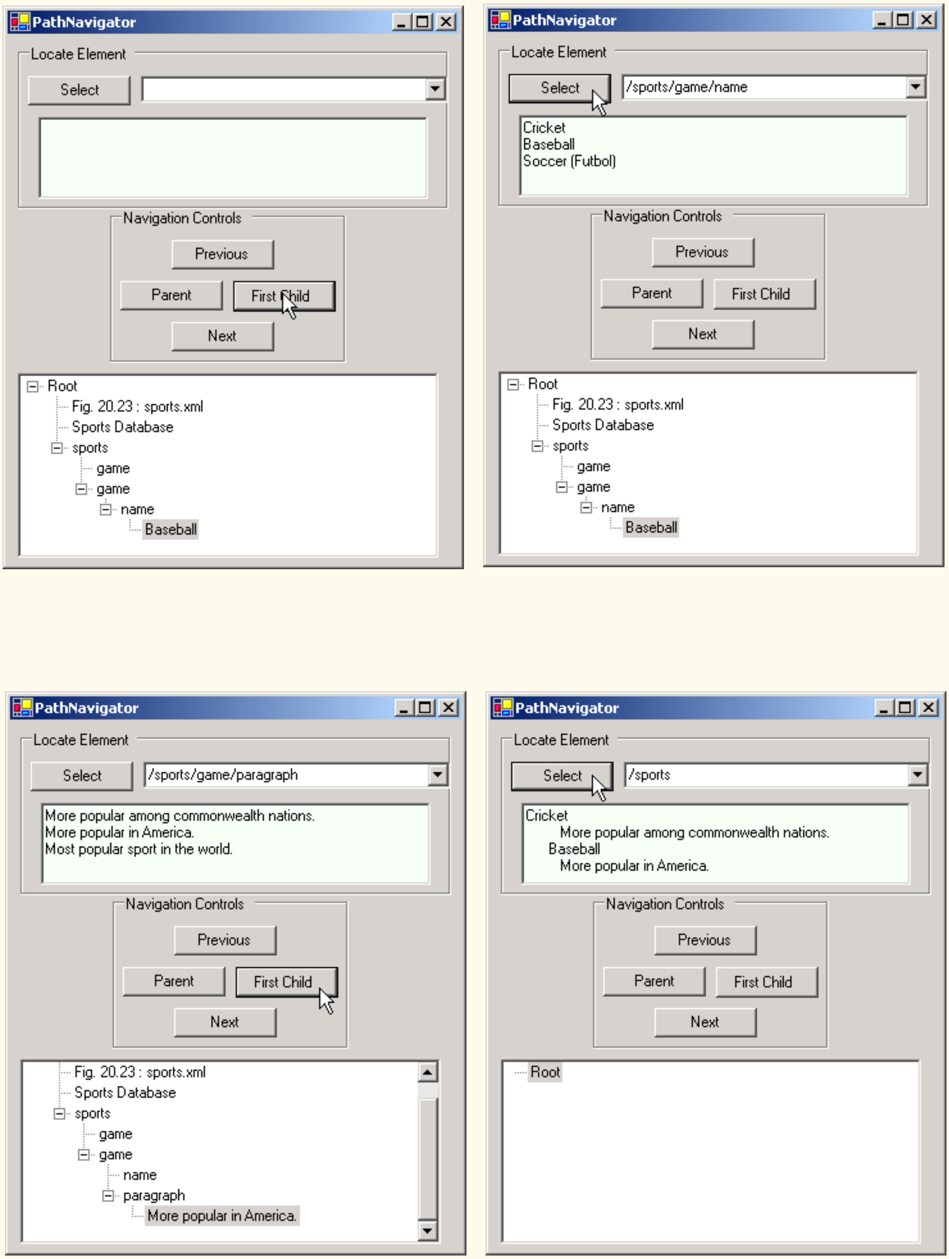
Fig. 18.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 3 of 7.)

```
            DetermineType( newNode, xpath );
            newTreeNode.Nodes.Add( newNode );

            // set current position for display
            tree = newNode;

            // update TreeView control
            pathTreeViewer.ExpandAll();
            pathTreeViewer.Refresh();
            pathTreeViewer.SelectedNode = tree;
         }
         else // node has no additional siblings
            MessageBox.Show( "Current node is last sibling.",
               "", MessageBoxButtons.OK,
               MessageBoxIcon.Information );
      } // end nextButton_Click

      // get previous sibling on previousButton click
      private void previousButton_Click( object sender,
         System.EventArgs e )
      {
         TreeNode parentTreeNode = null;

         // move to previous sibling
         if ( xpath.MoveToPrevious() )
         {
            parentTreeNode = tree.Parent; // get parent node

            // delete current node
            parentTreeNode.Nodes.Remove( tree );

            // move to previous node
            tree = parentTreeNode.LastNode;

            // update TreeView control
            pathTreeViewer.ExpandAll();
            pathTreeViewer.Refresh();
            pathTreeViewer.SelectedNode = tree;
         }
         else // if current node has no previous siblings
            MessageBox.Show( "Current node is first sibling.",
               "", MessageBoxButtons.OK,
               MessageBoxIcon.Information );
      } // end previousButton_Click

      // process selectButton click event
      private void selectButton_Click( object sender,
         System.EventArgs e )
      {
         XPathNodeIterator iterator; // enables node iteration

```

Fig. 18.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 4 of 7.)

```
         // get specified node from ComboBox
         try
         {
            iterator = xpath.Select( selectComboBox.Text );
            DisplayIterator( iterator ); // print selection
         }

         // catch invalid expressions
         catch ( System.ArgumentException argumentException )
         {
            MessageBox.Show( argumentException.Message,
               "Error", MessageBoxButtons.OK,
               MessageBoxIcon.Error );
         }
      } // end selectButton_Click

      // print values for XPathNodeIterator
      private void DisplayIterator( XPathNodeIterator iterator )
      {
         selectTreeViewer.Text = "";

         // prints selected node's values
         while ( iterator.MoveNext() )
            selectTreeViewer.Text +=
               iterator.Current.Value.Trim()
               + "\r\n";
      } // end DisplayIterator

      // determine if TreeNode should display current node
      // name or value
      private void DetermineType( TreeNode node,
         XPathNavigator xPath )
      {
         // determine NodeType
         switch ( xPath.NodeType )
         {
            // if Element, get its name
            case XPathNodeType.Element:

               // get current node name, and remove whitespace
               node.Text = xPath.Name.Trim();
               break;

            // obtain node values
            default:

               // get current node value and remove whitespace
               node.Text = xPath.Value.Trim();
               break;

         } // end switch
      } // end DetermineType
   } // end PathNavigator
```

Fig. 18.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 5 of 7.)


PathNavigator
Locate Element
Select
Navigation Controls
Previous
Parent
First Child
Next

```
Root
  Fig. 20.23 : sports.xml
  Sports Database
  sports
    game
    game
      name
        Baseball
```

/sports/game/name

```
Cricket
Baseball
Soccer (Futbol)
```

```
Root
  Fig. 20.23 : sports.xml
  Sports Database
  sports
    game
    game
      name
        Baseball
```

/sports/game/paragraph

```
More popular among commonwealth nations.
More popular in America.
Most popular sport in the world.
```

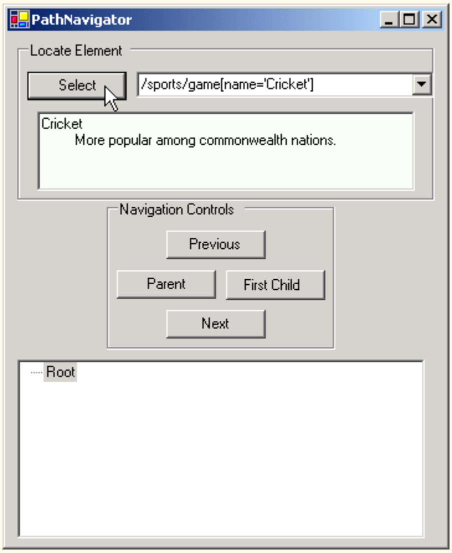
```
  Fig. 20.23 : sports.xml
  Sports Database
  sports
    game
    game
      name
      paragraph
        More popular in America.
```

/sports

```
Cricket
      More popular among commonwealth nations.
    Baseball
      More popular in America.
```

```
Root
```

Fig. 18.9 `XPathNavigator` class used to navigate selected nodes. (Part 6 of 7.)


PathNavigator
Locate Element
Select
Navigation Controls
Previous
Parent
First Child
Next

/sports/game[name='Cricket']

Cricket
More popular among commonwealth nations.

Root

Fig. 18.9 `XPathNavigator` class used to navigate selected nodes. (Part 7 of 7.)

```
<?xml version = "1.0"?>

<!-- Fig. 18.10: sports.xml -->
<!-- Sports Database        -->

<sports>

  <game id = "783">
      <name>Cricket</name>

      <paragraph>
         More popular among commonwealth nations.
      </paragraph>
   </game>

   <game id = "239">
      <name>Baseball</name>

      <paragraph>
         More popular in America.
      </paragraph>
   </game>

   <game id = "418">
      <name>Soccer(Futbol)</name>
      <paragraph>Most popular sport in the world</paragraph>
   </game>
</sports>
```

Fig. 18.10 XML document that describes various sports.

18.5 Document Type Definitions (DTDs), Schemas and Validation

XML documents can reference optional documents that specify how the XML documents should be structured. These optional documents are called *Document Type Definitions* (*DTDs)* and *Schemas*. When a DTD or Schema document is provided, some parsers (called *validating parsers)* can read the DTD or Schema and check the XML document's structure against it. If the XML document conforms to the DTD or Schema, then the XML document is *valid*. Parsers that cannot check for document conformity against the DTD or Schema are called *non-validating parsers*. If an XML parser (validating or non-validating) is able to process an XML document (that does not reference a DTD or Schema), the XML document is considered to be *well formed* (i.e., it is syntactically correct). By definition, a valid XML document is also a well-formed XML document. If a document is not well formed, parsing halts, and the parser issues an error.

Software Engineering Observation 18.2

DTD and Schema documents are essential components for XML documents used in business-to-business (B2B) transactions and mission-critical systems. These documents help ensure that XML documents are valid.

Software Engineering Observation 18.3

Because XML document content can be structured in many different ways, an application cannot determine whether the document data it receives is complete, missing data or ordered properly. DTDs and Schemas solve this problem by providing an extensible means of describing a document's contents. An application can use a DTD or Schema document to perform a validity check on the document's contents.

Expression	Description
`/sports`	Matches the `sports` node that is child node of the document root node. This node contains the root element.
`/sports/game/name`	Matches all `name` nodes that are child nodes of `game`. The `game` node must be a child of `sports` and `sports` must be a root element node.
`/sports/game/paragraph`	Matches all `paragraph` nodes that are child nodes of `game`. The `game` node must be a child of `sports`, and `sports` must be a root element node.
`/sports/game[name='Cricket']`	Matches all `game` nodes that contain a child element name whose value is `Cricket`. The `game` node must be a child of `sports`, and `sports` must be a root element node.

Fig. 18.11 XPath expressions and descriptions.

18.5.1 Document Type Definitions

Document type definitions (DTDs) provide a means for type checking XML documents and thus verifying their *validity* (confirming that elements contain the proper attributes, elements are in the proper sequence, etc.). DTDs use *EBNF* (*Extended Backus-Naur Form*) *grammar* to describe an XML document's content. XML parsers need additional functionality to read EBNF grammar, because it is not XML syntax. Although DTDs are optional, they are recommended to ensure document conformity. The DTD in Fig. 18.12 defines the set of rules (i.e., the grammar) for structuring the business letter document contained in Fig. 18.13.

Portability Tip 18.2

DTDs can ensure consistency among XML documents generated by different programs.

Line 4 uses the ***ELEMENT*** *element type declaration* to define rules for element **letter**. In this case, **letter** contains one or more **contact** elements, one **salutation** element, one or more **paragraph** elements, one **closing** element and one **signature** element, in that sequence. The *plus sign* (**+**) *occurrence indicator* specifies that an element must occur one or more times. Other indicators include the *asterisk* (*****), which indicates an optional element that can occur any number of times, and the *question mark* (**?**), which indicates an optional element that can occur at most once. If an occurrence indicator is omitted, exactly one occurrence is expected.

The **contact** element definition (line 7) specifies that it contains the **name**, **address1**, **address2**, **city**, **state**, **zip**, **phone** and **flag** elements—in that order. Exactly one occurrence of each is expected.

```
 1  <!-- Fig. 18.12: letter.dtd      -->
 2  <!-- DTD document for letter.xml -->
 3
 4  <!ELEMENT letter ( contact+, salutation, paragraph+,
 5     closing, signature )>
 6
 7  <!ELEMENT contact ( name, address1, address2, city, state,
 8     zip, phone, flag )>
 9  <!ATTLIST contact type CDATA #IMPLIED>
10
11  <!ELEMENT name ( #PCDATA )>
12  <!ELEMENT address1 ( #PCDATA )>
13  <!ELEMENT address2 ( #PCDATA )>
14  <!ELEMENT city ( #PCDATA )>
15  <!ELEMENT state ( #PCDATA )>
16  <!ELEMENT zip ( #PCDATA )>
17  <!ELEMENT phone ( #PCDATA )>
18  <!ELEMENT flag EMPTY>
19  <!ATTLIST flag gender (M | F) "M">
20
21  <!ELEMENT salutation ( #PCDATA )>
22  <!ELEMENT closing ( #PCDATA )>
23  <!ELEMENT paragraph ( #PCDATA )>
24  <!ELEMENT signature ( #PCDATA )>
```

Fig. 18.12 Document Type Definition (DTD) for a business letter.

Line 9 uses the ***`ATTLIST`*** *element type declaration* to define an attribute (i.e., **`type`**) for the **`contact`** element. Keyword ***`#IMPLIED`*** specifies that, if the parser finds a **`contact`** element without a **`type`** attribute, the application can provide a value or ignore the missing attribute. The absence of a **`type`** attribute cannot invalidate the document. Other types of default values include ***`#REQUIRED`*** and ***`#FIXED`***. Keyword **`#REQUIRED`** specifies that the attribute must be present in the document and the keyword **`#FIXED`** specifies that the attribute (if present) must always be assigned a specific value. For example,

```
<!ATTLIST address zip #FIXED "01757">
```

indicates that the value **`01757`** must be used for attribute **`zip`**; otherwise, the document is invalid. If the attribute is not present, then the parser, by default, uses the fixed value that is specified in the **`ATTLIST`** declaration. Flag ***`CDATA`*** specifies that attribute **`type`** contains a **`String`** that is not processed by the parser, but instead is passed to the application as is.

Software Engineering Observation 18.4

DTD syntax does not provide any mechanism for describing an element's (or attribute's) data type.

Flag ***`#PCDATA`*** (line 11) specifies that the element can store *parsed character data* (i.e., text). Parsed character data cannot contain markup. The characters less than (**`<`**) and ampersand (**`&`**) must be replaced by their *entities* (i.e., **`&lt;`** and **`&amp;`**). However, the ampersand character can be inserted when used with entities. See Appendix L (on CD) for a list of pre-defined entities.

Line 18 defines an empty element named **`flag`**. Keyword ***`EMPTY`*** specifies that the element cannot contain character data. Empty elements commonly are used for their attributes.

Common Programming Error 18.9

Any element, attribute or relationship not explicitly defined by a DTD results in an invalid document.

Many XML documents explicitly reference a DTD. Figure 18.13 is an XML document that conforms to **`letter.dtd`** (Fig. 18.12).

```
<?xml version = "1.0"?>

<!-- Fig. 18.13: letter2.xml             -->
<!-- Business letter formatted with XML -->

<!DOCTYPE letter SYSTEM "letter.dtd">

<letter>
   <contact type = "from">
      <name>Jane Doe</name>
      <address1>Box 12345</address1>
      <address2>15 Any Ave.</address2>
      <city>Othertown</city>
      <state>Otherstate</state>
      <zip>67890</zip>
      <phone>555-4321</phone>
```

Fig. 18.13 XML document referencing its associated DTD. (Part 1 of 2.)

```
17          <flag gender = "F" />
18       </contact>
19
20       <contact type = "to">
21          <name>John Doe</name>
22          <address1>123 Main St.</address1>
23          <address2></address2>
24          <city>Anytown</city>
25          <state>Anystate</state>
26          <zip>12345</zip>
27          <phone>555-1234</phone>
28          <flag gender = "M" />
29       </contact>
30
31       <salutation>Dear Sir:</salutation>
32
33          <paragraph>It is our privilege to inform you about our new
34          database managed with XML. This new system
35          allows you to reduce the load on your inventory list
36          server by having the client machine perform the work of
37          sorting and filtering the data.
38          </paragraph>
39
40          <paragraph>Please visit our Web site for availability
41          and pricing.
42          </paragraph>
43       <closing>Sincerely</closing>
44       <signature>Ms. Doe</signature>
45    </letter>
```

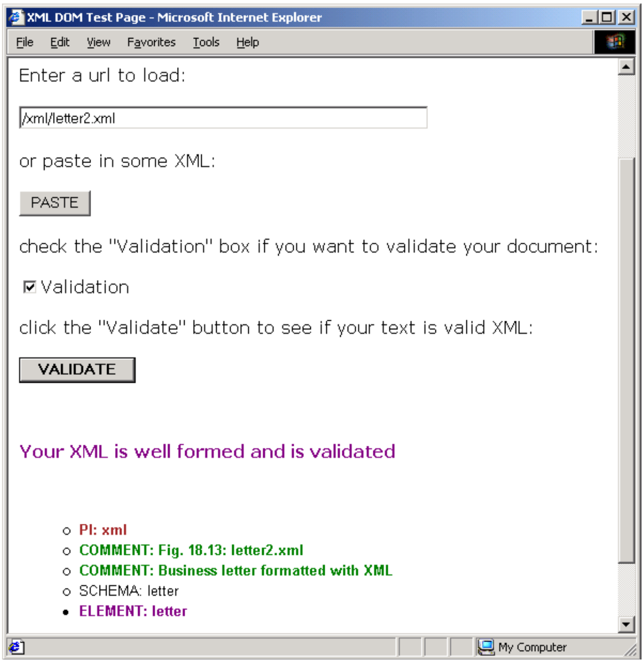
Fig. 18.13 XML document referencing its associated DTD. (Part 2 of 2.)

This XML document is similar to that in Fig. 18.3. Line 6 references a DTD file. This markup contains three pieces: The name of the root element (**`letter`** in line 8) to which the DTD is applied, the keyword ***`SYSTEM`*** (which in this case denotes an *external DTD*—a DTD defined in a separate file) and the DTD's name and location (i.e., **`letter.dtd`** in the current directory). Though almost any file extension can be used, DTD documents typically end with the ***`.dtd`*** extension.

Various tools (many of which are free) check document conformity against DTDs and Schemas (discussed momentarily). The output in Fig. 18.14 shows the results of the validation of **`letter2.xml`** using Microsoft's *XML Validator*. Visit **`www.w3.org/XML/Schema.html`** for a list of validating tools. Microsoft XML Validator is available free for download from

`msdn.microsoft.com/downloads/samples/Internet/xml/`
`xml_validator/sample.asp`

Microsoft XML Validator can validate XML documents against DTDs locally or by uploading the documents to the XML Validator Web site. Here, **`letter2.xml`** and **`letter.dtd`** are placed in folder **`C:\XML\`**. This XML document (**`letter2.xml`**) is well formed and conforms to **`letter.dtd`**.


XML DOM Test Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
PASTE
VALIDATE
My Computer

Enter a url to load:

/xml/letter2.xml

or paste in some XML:

check the "Validation" box if you want to validate your document:

☑ Validation

click the "Validate" button to see if your text is valid XML:

Your XML is well formed and is validated

- PI: xml
- COMMENT: Fig. 18.13: letter2.xml
- COMMENT: Business letter formatted with XML
- SCHEMA: letter
- ELEMENT: letter

Fig. 18.14 XML Validator validates an XML document against a DTD.

XML documents that fail validation are still well-formed documents. When a document fails to conform to a DTD or Schema, Microsoft XML Validator displays an error message. For example, the DTD in Fig. 18.12 indicates that the **`contacts`** element must contain child element **`name`**. If the document omits this child element, the document is well formed, but not valid. In such a scenario, Microsoft XML Validator displays the error message shown in Fig. 18.15.

C# programs can use msxml to validate XML documents against DTDs. For information on how to accomplish this, visit:

`msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguidnf/html/cpconvalidationagainstdtdwithxmlvalidatingreader.asp`

Schemas are the preferred means of defining structures for XML documents in .NET. Although, several types of Schemas exist, the two most popular are Microsoft Schema and W3C Schema. We begin our discussion of Schemas in the next section.

18.5.2 Microsoft XML Schemas[2]

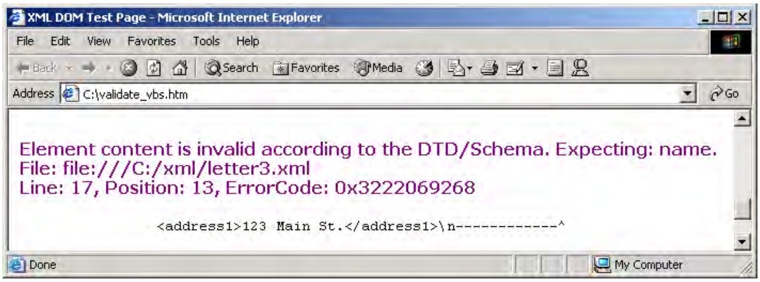
In this section, we introduce an alternative to DTDs—called Schemas—for defining an XML document's structure. Many developers in the XML community feel that DTDs are not flexible enough to meet today's programming needs. For example, DTDs cannot be manipulated (e.g., searched, programmatically modified, etc.) in the same manner that XML documents can, because DTDs are not XML documents. Furthermore, DTDs do not provide features for describing an element's (or attribute's) data type.

---

[2] W3C Schema, which we discuss in Section 18.5.3, is emerging as the industry standard for describing an XML document's structure. Within the next two years, we expect most developers will be using W3C Schema.

Unlike DTDs, Schemas do not use Extended Backus-Naur Form (EBNF) grammar. Instead, Schemas are XML documents that can be manipulated (e.g., elements can be added or removed, etc.) like any other XML document. As with DTDs, Schemas require validating parsers.

In this section, we focus on Microsoft's *XML Schema* vocabulary. Figure 18.16 presents an XML document that conforms to the Microsoft Schema document shown in Fig. 18.17. By convention, Microsoft XML Schema documents use the file extension ***.xdr***, which is short for *XML-Data Reduced*. Line 6 (Fig. 18.16) references the Schema document **`book.xdr`**.


XML DOM Test Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\validate_vbs.htm
Go
Done
My Computer

```
Element content is invalid according to the DTD/Schema. Expecting: name.
File: file:///C:/xml/letter3.xml
Line: 17, Position: 13, ErrorCode: 0x3222069268

<address1>123 Main St.</address1>\n------------^
```

Fig. 18.15 XML Validator displaying an error message.

```
1  <?xml version = "1.0"?>
2  
3  <!-- Fig. 18.16: bookxdr.xml          -->
4  <!-- XML file that marks up book data -->
5  
6  <books xmlns = "x-schema:book.xdr">
7     <book>
8        <title>C# How to Program</title>
9     </book>
10 
11    <book>
12       <title>Java How to Program, 4/e</title>
13    </book>
14 
15    <book>
16       <title>Visual Basic .NET How to Program</title>
17    </book>
18 
19    <book>
20       <title>Advanced Java 2 Platform How to Program</title>
21    </book>
```

Fig. 18.16 XML document that conforms to a Microsoft Schema document. (Part 1 of 2.)

```

   <book>
      <title>Python How to Program</title>
   </book>
</books>
```

Fig. 18.16 XML document that conforms to a Microsoft Schema document. (Part 2 of 2.)

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 18.17: book.xdr                       -->
4  <!-- Schema document to which book.xml conforms -->
5
6  <Schema xmlns = "urn:schemas-microsoft-com:xml-data">
7     <ElementType name = "title" content = "textOnly"
8        model = "closed" />
9
10    <ElementType name = "book" content = "eltOnly" model = "closed">
11       <element type = "title" minOccurs = "1" maxOccurs = "1" />
12    </ElementType>
13
14    <ElementType name = "books" content = "eltOnly" model = "closed">
15       <element type = "book" minOccurs = "0" maxOccurs = "*" />
16    </ElementType>
17 </Schema>
```

Fig. 18.17 Microsoft Schema file that contains structure to which `bookxdr.xml` conforms.

Software Engineering Observation 18.5

Schemas are XML documents that conform to DTDs, which define the structure of a Schema. These DTDs, which are bundled with the parser, are used to validate the Schemas that authors create.

Software Engineering Observation 18.6

Many organizations and individuals are creating DTDs and Schemas for a broad range of categories (e.g., financial transactions, medical prescriptions, etc.). Often, these collections—called repositories*—are available free for download from the Web.*[3]

In line 6, root element ***`Schema`*** begins the Schema markup. Microsoft Schemas use the namespace URI ***`"urn:schemas-microsoft-com:data"`***. Line 7 uses element ***`ElementType`*** to define element **`title`**. Attribute ***`content`*** specifies that this element contains parsed character data (i.e., text only). Element **`title`** is not permitted to contain child elements. Setting the ***`model`*** *attribute* to ***`"closed"`*** specifies that a conforming XML document can contain only elements defined in this Schema. Line 10 defines element **`book`**; this element's **`content`** is "elements only" (i.e., ***`eltOnly`***). This means that the element cannot contain mixed content (i.e., text and other elements). Within the **`ElementType`** element named **`book`**, the ***`element`*** element indicates that **`title`** is a **`child`** element of **`book`**. Attributes ***`minOccurs`*** and ***`maxOccurs`*** are set to **`"1"`**, indicating that a **`book`** element must contain exactly one **`title`** element. The asterisk (**`*`**) in line 15 indicates that the Schema permits any number of **`book`** elements in element **`books`**. We discuss how to validate **`bookxdr.xml`** against **`book.xdr`** in Section 18.5.4.

---

[3] See, for example, **`opengis.net/schema.htm`**.

18.5.3 W3C XML Schema[4]

In this section, we focus on *W3C XML Schema*[5]—the schema that the W3C created. XML Schema is a *Recommendation* (i.e., a stable release suitable for use in industry). Figure 18.18 shows a Schema-valid XML document named **`bookxsd.xml`** and Fig. 18.19 shows the W3C XML Schema document (**`book.xsd`**) that defines the structure for **`bookxsd.xml`**. Although Schema authors can use virtually any filename extension, W3C XML Schemas typically use the ***`.xsd`*** extension. We discuss how to validate **`bookxsd.xml`** against **`book.xsd`** in the next section.

```
<?xml version = "1.0"?>

<!-- Fig. 18.18: bookxsd.xml                  -->
<!-- Document that conforms to W3C XML Schema -->

<deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
   <book>
      <title>e-Business and e-Commerce How to Program</title>
   </book>
   <book>
      <title>Python How to Program</title>
   </book>
</deitel:books>
```

Fig. 18.18 XML document that conforms to W3C XML Schema.

```
<?xml version = "1.0"?>

<!-- Fig. 18.19: book.xsd           -->
<!-- Simple W3C XML Schema document -->

<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
   xmlns:deitel = "http://www.deitel.com/booklist"
   targetNamespace = "http://www.deitel.com/booklist">

   <xsd:element name = "books" type = "deitel:BooksType"/>

   <xsd:complexType name = "BooksType">
      <xsd:sequence>
         <xsd:element name = "book" type = "deitel:BookType"
         minOccurs = "1" maxOccurs = "unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
```

Fig. 18.19 XSD Schema document to which **`bookxsd.xml`** conforms.

---

[4] We provide a detailed treatment of W3C Schema in *XML How to Program, 2/e*.

[5] For the latest on W3C XML Schema, visit **`www.w3.org/XML/Schema`**.

```
18
19    <xsd:complexType name = "BookType">
20       <xsd:sequence>
21          <xsd:element name = "title" type = "xsd:string"/>
22       </xsd:sequence>
23    </xsd:complexType>
24
25 </xsd:schema>
```

Fig. 18.19 XSD Schema document to which **`bookxsd.xml`** conforms.

W3C XML Schema use the namespace URI ***`http://www.w3.org/2001/XMLSchema`*** and often use *namespace prefix* ***`xsd`*** (line 6 in Fig. 18.19). Root element ***`schema`*** contains elements that define the XML document's structure. Line 7 binds the URI **`http://www.deitel.com/booklist`** to namespace prefix **`deitel`**. Line 8 specifies the ***`targetNamespace`***, which is the namespace for elements and attributes that this schema defines.

In W3C XML Schema, element ***`element`*** (line 10) defines an element. Attributes ***`name`*** and ***`type`*** specify the **`element`**'s name and data type, respectively. In this case, the name of the element is **`books`** and the data type is **`deitel:BooksType`**. Any element (e.g., **`books`**) that contains attributes or child elements must define a *complex type*, which defines each attribute and child element. Type **`deitel:BooksType`** (lines 12–17) is an example of a complex type. We prefix **`BooksType`** with **`deitel`**, because this is a complex type that we have created, not an existing W3C XML Schema complex type.

Lines 12–17 use element ***`complexType`*** to define an element type that has a child element named **`book`**. Because **`book`** contains a child element, its type must be a complex type (e.g., **`BookType`**). Attribute ***`minOccurs`*** specifies that **`books`** must contain a minimum of one **`book`** element. Attribute ***`maxOccurs`***, with value ***`unbounded`*** (line 14) specifies that **`books`** may have any number of **`book`** child elements. Element ***`sequence`*** specifies the order of elements in the complex type.

Lines 19–23 define the **`complexType BookType`**. Line 21 defines element **`title`** with **`type`** ***`xsd:string`***. When an element has a *simple type* such as **`xsd:string`**, it is prohibited from containing attributes and child elements. W3C XML Schema provides a large number of data types such as ***`xsd:date`*** for dates, ***`xsd:int`*** for integers, ***`xsd:double`*** for floating-point numbers and ***`xsd:time`*** for time.

Good Programming Practice 18.1

By convention, W3C XML Schema authors use namespace prefix **`xsd`** *when referring to the URI* ***`http://www.w3.org/2001/XMLSchema`***.

18.5.4 Schema Validation in C#

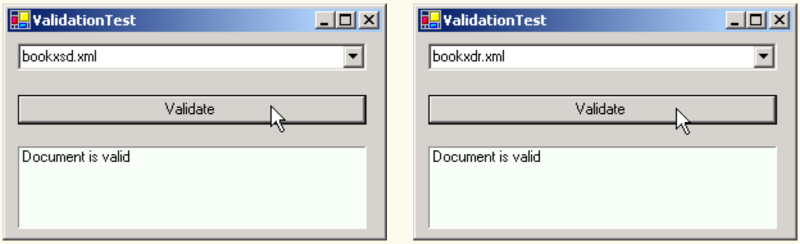
In this section, we present a C# application (Fig. 18.20) that uses classes from the .NET Framework Class Library to validate the XML documents presented in the last two sections against their respective Schemas. We use an instance of ***`XmlValidatingReader`*** to perform the validation.

Line 17 creates an ***`XmlSchemaCollection`*** reference named **`schemas`**. Line 28 calls method ***`Add`*** to add an ***`XmlSchema`*** object to the Schema collection. Method **`Add`** is passed a name that identifies the Schema (i.e., **`"book"`**) and the name of the Schema file (i.e., **"book.xdr"**). Line 29 calls method **Add** to add a W3C XML Schema. The first argument specifies the namespace URI (i.e., line 18 in Fig. 18.19) and the second argument indentifies the schema file (i.e., **"book.xsd"**). This is the Schema that is used to validate **bookxsd.xml**.

```
 1  // Fig. 18.20: ValidationTest.cs
 2  // Validating XML documents against Schemas.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml;
 7  using System.Xml.Schema;         // contains Schema classes
 8
 9  // determines XML document Schema validity
10  public class ValidationTest : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.ComboBox filesComboBox;
13     private System.Windows.Forms.Button validateButton;
14     private System.Windows.Forms.Label consoleLabel;
15     private System.ComponentModel.Container components = null;
16
17     private XmlSchemaCollection schemas;   // Schemas
18     private bool valid;                    // validation result
19
20     public ValidationTest()
21     {
22        InitializeComponent();
23
24        valid = true;  // assume document is valid
25
26        // get Schema(s) for validation
27        schemas = new XmlSchemaCollection();
28        schemas.Add( "book", "book.xdr" );
29        schemas.Add( "http://www.deitel.com/booklist", "book.xsd" );
30     } // end constructor
31
32     // Visual Studio .NET generated code
33
34     [STAThread]
35     static void Main()
36     {
37        Application.Run( new ValidationTest() );
38     } // end Main
39
40     // handle validateButton click event
41     private void validateButton_Click( object sender,
42        System.EventArgs e )
43     {
44        // get XML document
45        XmlTextReader reader =
46           new XmlTextReader( filesComboBox.Text );
47
```

Fig. 18.20 Schema-validation example. (Part 1 of 2.)

```
48       // get validator
49       XmlValidatingReader validator =
50          new XmlValidatingReader( reader );
51
52       // assign Schema(s)
53       validator.Schemas.Add( schemas );
54
55       // set validation type
56       validator.ValidationType = ValidationType.Auto;
57
58       // register event handler for validation error(s)
59       validator.ValidationEventHandler +=
60          new ValidationEventHandler( ValidationError );
61
62       // validate document node-by-node
63       while ( validator.Read() ) ; // empty body
64
65       // check validation result
66       if ( valid )
67          consoleLabel.Text = "Document is valid";
68
69       valid = true; // reset variable
70
71       // close reader stream
72       validator.Close();
73    }  // end validateButton_Click
74
75    // event handler for validation error
76    private void ValidationError( object sender,
77       ValidationEventArgs arguments )
78    {
79       consoleLabel.Text = arguments.Message;
80       valid = false; // validation failed
81    } // end ValidationError
82 } // end ValidationTest
```



bookxsd.xml

Document is valid

bookxdr.xml

Document is valid

Fig. 18.20 Schema-validation example. (Part 2 of 2.)

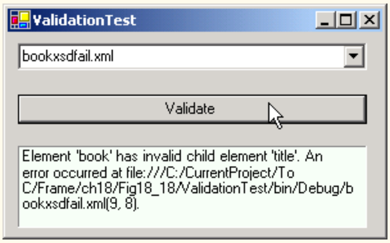
Lines 45–46 create an **`XmlReader`** for the file that the user selected from **`filesComboBox`**. The XML document to be validated against a Schema contained in the **`XmlSchemaCollection`** must be passed to the **`XmlValidatingReader`** constructor (lines 49–50).

Line 53 **`Add`**s the Schema collection referenced by **`Schemas`** to the ***`Schemas`*** *property*. This property sets the Schema used to validate the document. The ***`ValidationType`*** property (line 56) is set to the ***`ValidationType`*** *enumeration* constant for **`Auto`**matically identifying the Schema's type (i.e., XDR or XSD). Lines 59–60 register method **`ValidationError`** with ***`ValidationEventHandler`***. Method **`ValidationError`** (lines 76–81) is called if the document is invalid or an error occurs, such as if the document cannot be found. Failure to register a method with **`ValidationEventHandler`** causes an exception to be thrown when the document is missing or invalid.

Validation is performed node-by-node by calling the method ***`Read`*** (line 63). Each call to **`Read`** validates the next node in the document. The loop terminates either when all nodes have been validated successfully or a node fails validation. When validated against their respective Schemas, the XML documents in Fig. 18.16 and Fig. 18.18 validate successfully.

Figure 18.21 and Fig. 18.22 list two XML documents that fail to conform to **`book.xdr`** and **`book.xsd`**, respectively. In Fig. 18.21, the extra **`title`** element in **`book`** (lines 19–22) invalidate the document. In Fig. 18.22, the extra **`title`** element in **`book`** (lines 7–10) invalidates the document. Although both documents are invalid, they are well formed.

```
1   <?xml version = "1.0"?>
2
3  <!-- Fig. 18.22: bookxsdfail.xml                  -->
4  <!-- Document that does not conforms to W3C Schema -->
5
6  <deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
7     <book>
8        <title>e-Business and e-Commerce How to Program</title>
9        <title>C# How to Program</title>
10    </book>
11    <book>
12       <title>Python How to Program</title>
13    </book>
14 </deitel:books>
```



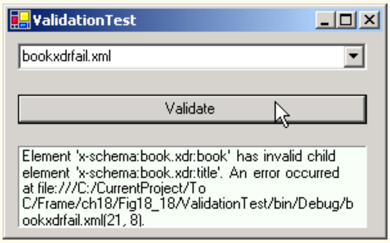
```
Element 'book' has invalid child element 'title'. An
error occurred at file:///C:/CurrentProject/To
C/Frame/ch18/Fig18_18/ValidationTest/bin/Debug/b
ookxsdfail.xml(9, 8).
```

Fig. 18.21 XML document that does not conform to the XSD schema of Fig. 18.19.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 18.22: bookxdrfail.xml                      -->
4  <!-- XML file that does not conform to Schema book.xdr -->
5
```

Fig. 18.22 XML file that does not conform to the Schema in Fig. 18.17. (Part 1 of 2.)

```
<books xmlns = "x-schema:book.xdr">
   <book>
      <title>XML How to Program</title>
   </book>

   <book>
      <title>Java How to Program, 4/e</title>
   </book>

   <book>
      <title>Visual Basic .NET How to Program</title>
   </book>

   <book>
      <title>C++ How to Program, 3/e</title>
      <title>Python How to Program</title>
   </book>

   <book>
      <title>C# How to Program</title>
   </book>
</books>
```



```
Element 'x-schema:book.xdr:book' has invalid child
element 'x-schema:book.xdr:title'. An error occurred
at file:///C:/CurrentProject/To
C/Frame/ch18/Fig18_18/ValidationTest/bin/Debug/b
ookxdrfail.xml(21, 8).
```

Fig. 18.22 XML file that does not conform to the Schema in Fig. 18.17. (Part 2 of 2.)

18.6 Extensible Stylesheet Language and `XslTransform`

Extensible Stylesheet Language (XSL) is an XML vocabulary for formatting XML data. In this section, we discuss the portion of XSL—called *XSL Transformations* (*XSLT*)—that creates formatted text-based documents from XML documents. This process is called a *transformation* and involves two tree structures: The *source tree*, which is the XML document being transformed, and the *result tree*, which is the result (i.e., any text-based format such as XHTML) of the transformation.[6] The source tree is not modified when a transformation occurs.

To perform transformations, an XSLT processor is required. Popular XSLT processors include Microsoft's msxml and the Apache Software Foundation's *Xalan*. The XML document, shown in Fig. 18.23, is transformed by msxml into an XHTML document (Fig. 18.24).

---

[6] Extensible Hypertext Markup Language (XHTML) is the W3C technical recommendation that replaces HTML for marking up content for the Web. For more information on XHTML, see the XHTML Appendices K and L on the CD and visit **`www.w3.org`**.

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 18.23: sorting.xml                  -->
 4  <!-- XML document containing book information -->
 5
 6  <?xml:stylesheet type = "text/xsl" href = "sorting.xsl"?>
 7
 8  <book isbn = "999-99999-9-X">
 9     <title>Deitel&apos;s XML Primer</title>
10
11     <author>
12        <firstName>Paul</firstName>
13        <lastName>Deitel</lastName>
14     </author>
15
16     <chapters>
17        <frontMatter>
18           <preface pages = "2" />
19           <contents pages = "5" />
20           <illustrations pages = "4" />
21        </frontMatter>
22
23        <chapter number = "3" pages = "44">
24           Advanced XML</chapter>
25
26        <chapter number = "2" pages = "35">
27           Intermediate XML</chapter>
28
29        <appendix number = "B" pages = "26">
30           Parsers and Tools</appendix>
31
32        <appendix number = "A" pages = "7">
33           Entities</appendix>
34
35        <chapter number = "1" pages = "28">
36           XML Fundamentals</chapter>
37     </chapters>
38
39     <media type = "CD" />
40  </book>
```

Fig. 18.23 XML document containing book information.

Line 6 is a *processing instruction* (*PI*), which contains application-specific information that is embedded into the XML document. In this particular case, the processing instruction is specific to IE and specifies the location of an XSLT document with which to transform the XML document. The characters **`<?`** and **`?>`** delimit a processing instruction, which consists of a *PI target* (e.g., **`xml:stylesheet`**) and *PI value* (e.g., **`type = "text/xsl" href = "sorting.xsl"`**). The portion of this particular PI value that follows **`href`** specifies the name and location of the style sheet to apply—in this case, **`sorting.xsl`**, which is located in the same directory as this XML document.

Fig. 18.24 presents the XSLT document (**`sorting.xsl`**) that transforms **`sorting.xml`** (Fig. 18.23) to XHTML.

Performance Tip 18.1

Using Internet Explorer on the client to process XSLT documents conserves server resources by using the client's processing power (instead of having the server process XSLT documents for multiple clients).

Line 1 of Fig. 18.23 contains the XML declaration. Recall that an XSL document is an XML document. Line 6 is the **`xsl:stylesheet`** root element. Attribute ***`version`*** specifies the version of XSLT to which this document conforms. Namespace prefix **`xsl`** is defined and is bound to the XSLT URI defined by the W3C. When processed, lines 11–13 write the document type declaration to the result tree. Attribute **`method`** is assigned **`"xml"`**, which indicates that XML is being output to the result tree. Attribute **`omit-xml-declaration`** is assigned **`"no"`**, which outputs an XML declaration to the result tree. Attribute **`doctype-system`** and **`doctype-public`** write the **`Doctype`** DTD information to the result tree.

XSLT documents contain one or more ***`xsl:template`*** elements that specify which information is output to the result tree. The template on line 16 ***`match`***es the source tree's document root. When the document root is encountered, this template is applied, and any text marked up by this element that is not in the namespace referenced by **`xsl`** is output to the result tree. Line 18 calls for all the **`template`**s that match children of the document root to be applied. Line 23 specifies a **`template`** that **`match`**es element **`book`**.

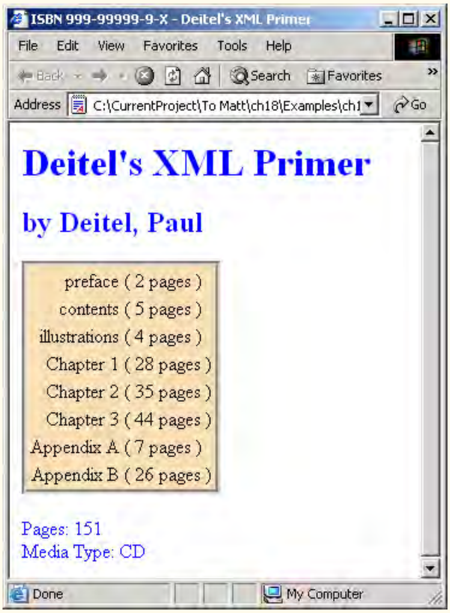
```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 18.24: sorting.xsl                       -->
4  <!-- Transformation of book information into XHTML -->
5
6  <xsl:stylesheet version = "1.0"
7     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9     <!-- write XML declaration and DOCTYPE DTD information -->
10    <xsl:output method = "xml" omit-xml-declaration = "no"
11       doctype-system =
12          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
13       doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
14
15    <!-- match document root -->
16    <xsl:template match = "/">
17       <html xmlns = "http://www.w3.org/1999/xhtml">
18          <xsl:apply-templates/>
19       </html>
20    </xsl:template>
21
22    <!-- match book -->
23    <xsl:template match = "book">
24       <head>
25          <title>ISBN <xsl:value-of select = "@isbn" /> -
26             <xsl:value-of select = "title" /></title>
27       </head>
```

Fig. 18.24 XSL document that transforms **`sorting.xml`** (Fig. 18.23) into XHTML. (Part 1 of 3.)

```

      <body>
         <h1 style = "color: blue">
            <xsl:value-of select = "title"/></h1>

         <h2 style = "color: blue">by <xsl:value-of
            select = "author/lastName" />,
            <xsl:value-of select = "author/firstName" /></h2>

         <table style =
            "border-style: groove; background-color: wheat">

            <xsl:for-each select = "chapters/frontMatter/*">
               <tr>
                  <td style = "text-align: right">
                     <xsl:value-of select = "name()" />
                  </td>

                  <td>
                    ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/chapter">
               <xsl:sort select = "@number" data-type = "number"
                    order = "ascending" />
               <tr>
                  <td style = "text-align: right">
                     Chapter <xsl:value-of select = "@number" />
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/appendix">
               <xsl:sort select = "@number" data-type = "text"
                    order = "ascending" />
               <tr>
                  <td style = "text-align: right">
                     Appendix <xsl:value-of select = "@number" />
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>
         </table>
```

Fig. 18.24 XSL document that transforms `sorting.xml` (Fig. 18.23) into XHTML. (Part 2 of 3.)

```
80
81             <br /><p style = "color: blue">Pages:
82                <xsl:variable name = "pagecount"
83                   select = "sum(chapters//*/@pages)" />
84                <xsl:value-of select = "$pagecount" />
85             <br />Media Type:
86                <xsl:value-of select = "media/@type" /></p>
87          </body>
88       </xsl:template>
89
90    </xsl:stylesheet>
```


ISBN 999-99999-9-X - Deitel's XML Primer
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\CurrentProject\To Matt\ch18\Examples\ch1
Go
Done
My Computer

Deitel's XML Primer

by Deitel, Paul

preface (2 pages)
contents (5 pages)
illustrations (4 pages)
Chapter 1 (28 pages)
Chapter 2 (35 pages)
Chapter 3 (44 pages)
Appendix A (7 pages)
Appendix B (26 pages)

Pages: 151
Media Type: CD

Fig. 18.24 XSL document that transforms `sorting.xml` (Fig. 18.23) into XHTML. (Part 3 of 3.)

Lines 25–26 create the title for the XHTML document. We use the ISBN of the book from attribute `isbn` and the contents of element `title` to create the title string **ISBN 999-99999-9-X - Deitel's XML Primer**. Element `xsl:value-of` selects the `book` element's `isbn` attribute.

Lines 33–35 create a header element that contains the book's author. Because the *context node* (i.e., the current node being processed) is `book`, the XPath expression `author/lastName` selects the author's last name, and the expression `author/firstName` selects the author's first name.

Line 40 selects each element (indicated by an asterisk) that is a child of element `frontMatter`. Line 43 calls *node-set function* ***`name`*** to retrieve the current node's element name (e.g., `preface`). The current node is the context node specified in the `xsl:for-each` (line 40).

Lines 53–54 sort `chapters` by number in ascending order. Attribute `select` selects the value of context node `chapter`'s attribute `number`. Attribute ***`data-type`*** with value `"number"`, specifies a numeric sort and attribute *`order`* specifies `"ascending"` order. Attribute `data-type` also can, be assigned the value *`"text"`* (line 67) and attribute `order` also may be assigned the value *`"descending"`*.

Lines 82–83 use an *XSL variable* to store the value of the book's page count and output it to the result tree. Attribute `name` specifies the variable's name, and attribute `select` assigns it a value. Function *`sum`* totals the values for all `page` attribute values. The two slashes between `chapters` and `*` indicate that all descendent nodes of `chapters` are searched for elements that contain an attribute named `pages`.

The *`System.Xml.Xsl`* namespace provides classes for applying XSLT style sheets to XML documents. Specifically, an object of class *`XslTransform`* performs the transformation.

Figure 18.25 applies a style sheet (`sports.xsl`) to `sports.xml` (Fig. 18.10). The transformation result is written to a text box and to a file. We also show the transformation results rendered in IE.

Line 20 declares `XslTransform` reference `transformer`. An object of this type is necessary to transform the XML data to another format. In line 29, the XML document is parsed and loaded into memory with a call to method `Load`. Method `CreateNavigator` is called in line 32 to create an `XPathNavigator` object, which is used to navigate the XML document during the transformation. A call to method *`Load`* of class `XslTransform` (line 36) parses and loads the style sheet that this application uses. The argument that is passed contains the name and location of the style sheet.

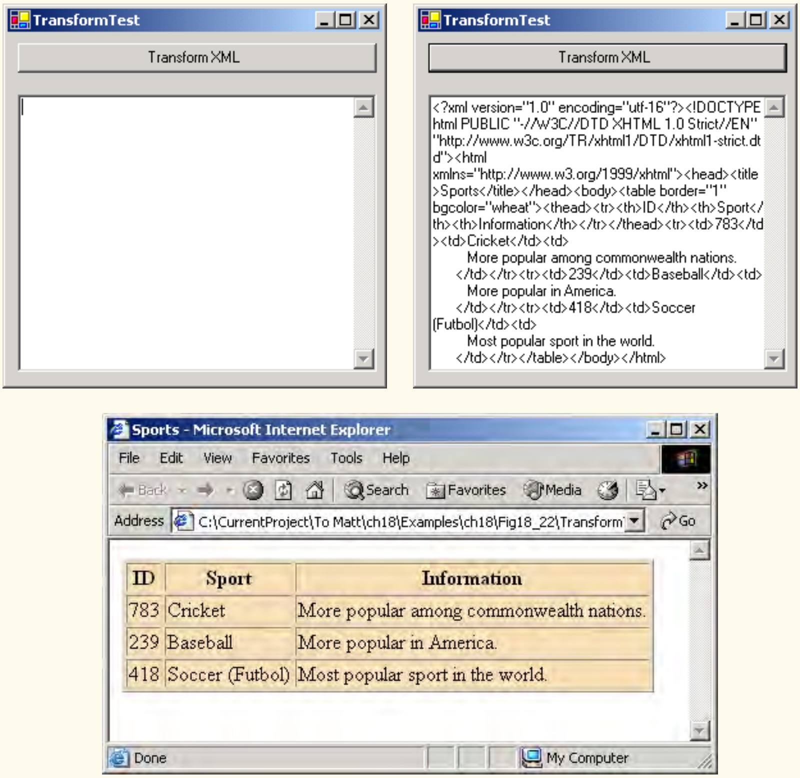
Event handler `transformButton_Click` calls method *`Transform`* of class `XslTransform` to apply the style sheet (`sports.xsl`) to `sports.xml` (line 53). This method takes three arguments: An `XPathNavigator` (created from `sports.xml`'s `XmlDocument`), an instance of class *`XsltArgumentList`*, which is a list of `string` parameters that can be applied to a style sheet—`null`, in this case and an instance of a derived class of `TextWriter` (in this example, an instance of class `StringWriter`). The results of the transformation are stored in the `StringWriter` object referenced by `output`. Lines 59–62 write the transformation results to disk. The third screen shot depicts the created XHTML document when it is rendered in IE.

```
 1  // Fig. 18.25: TransformTest.cs
 2  // Applying a style sheet to an XML document.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml;
 7  using System.Xml.XPath;   // contains XPath classes
 8  using System.Xml.Xsl;     // contains style sheet classes
 9  using System.IO;          // contains stream classes
10
11  // transforms XML document to XHTML
12  public class TransformTest : System.Windows.Forms.Form
13  {
14     private System.Windows.Forms.TextBox consoleTextBox;
15     private System.Windows.Forms.Button transformButton;
16     private System.ComponentModel.Container components = null;
```

Fig. 18.25 XSL style sheet applied to an XML document. (Part 1 of 3.)

```

   private XmlDocument document;     // Xml document root
   private XPathNavigator navigator; // navigate document
   private XslTransform transformer; // transform document
   private StringWriter output;      // display document

   public TransformTest()
   {
      InitializeComponent();

      // load XML data
      document = new XmlDocument();
      document.Load( "..\\..\\sports.xml" );

      // create navigator
      navigator = document.CreateNavigator();

      // load style sheet
      transformer = new XslTransform();
      transformer.Load( "..\\..\\sports.xsl" );
   } // end constructor

   // Windows Form Designer generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new TransformTest() );
   } // end Main

   // transformButton click event
   private void transformButton_Click( object sender,
      System.EventArgs e )
   {
      // transform XML data
      output = new StringWriter();
      transformer.Transform( navigator, null, output );

      // display transformation in text box
      consoleTextBox.Text = output.ToString();

      // write transformation result to disk
      FileStream stream = new FileStream( "..\\..\\sports.html",
         FileMode.Create );
      StreamWriter writer = new StreamWriter( stream );
      writer.Write( output.ToString() );

      // close streams
      writer.Close();
      output.Close();
   } // end transformButton_Click
}    // end TransformTest
```

Fig. 18.25 XSL style sheet applied to an XML document. (Part 2 of 3.)


TransformTest
Transform XML
TransformTest
Transform XML
Sports - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_22\Transform
Go
Done
My Computer

```
<?xml version="1.0" encoding="utf-16"?><!DOCTYPE
html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3c.org/TR/xhtml1/DTD/xhtml1-strict.dt
d"><html
xmlns="http://www.w3.org/1999/xhtml"><head><title
>Sports</title></head><body><table border="1"
bgcolor="wheat"><thead><tr><th>ID</th><th>Sport</
th><th>Information</th></tr></thead><tr><td>783</td
><td>Cricket</td><td>
        More popular among commonwealth nations.
     </td></tr><tr><td>239</td><td>Baseball</td><td>
        More popular in America.
     </td></tr><tr><td>418</td><td>Soccer
(Futbol)</td><td>
        Most popular sport in the world.
     </td></tr></table></body></html>
```

ID	Sport	Information
783	Cricket	More popular among commonwealth nations.
239	Baseball	More popular in America.
418	Soccer (Futbol)	Most popular sport in the world.

Fig. 18.25 XSL style sheet applied to an XML document. (Part 3 of 3.)

18.7 Microsoft BizTalk™

Increasingly, organizations are using the Internet to exchange critical data between business partners and their own business divisions. However, transferring data between organizations can become difficult, because companies often use different platforms, applications and data specifications that complicate data transfer. For example, consider a business that supplies raw materials to a variety of industries. If the supplier cannot receive all orders electronically because their customers use different computing platforms, an employee must input order data manually. If the supplier receives hundreds of orders a day, typing mistakes are likely, resulting in incorrect inventories or wrong order fulfillments, thereby jeorpardizing the business by losing customers.

The supplier has several options—either continue to have data entered manually, purchase the same software packages as the ones their customers use or encourage customers to adopt the applications used by the supply company. In a growing economy, a business would have to purchase and maintain disparate software packages, spend money for more employees to process data or force their business partners to standardize their own organizational software programs. To facilitate the flow of information between businesses, Microsoft developed *BizTalk* ("business talk"), an XML-based technology that helps to manage and facilitate business transactions.

BizTalk creates an environment in which data marked up as XML is used to exchange business-specific information, regardless of platform or programming applications. This section overviews BizTalk and presents a code example to illustrate the business-specific information included in the markup.

BizTalk consists of three parts: The BizTalk Server, the BizTalk Framework and the BizTalk Schema Library. The *BizTalk Server* (*BTS*) parses and translates all inbound and outbound messages (or documents) that are sent to and from a business, using Internet standards such as HTTP. The *BizTalk Framework* is a Schema for structuring those messages. The Framework offers a specific set of core tags. Businesses can download the Framework to use in their organizations and can submit new schemas to the BizTalk organization, at **`www.biztalk.org`**. Once the BizTalk organization verifies and validates the submissions, the Schemas become BizTalk Framework Schemas. The *BizTalk Schema Library* is a collection of Framework Schemas. Figure 18.26 summarizes BizTalk terminology.

Fig. 18.27 is an example BizTalk message for a product offer from a clothing company. The message Schema for this example was developed by Microsoft to facilitate online purchases by a retailer from a wholesaler. We use this Schema for a fictitious company, named ExComp.

BizTalk	**Description**
Framework	A specification that defines a format for messages.
Schema library	A repository of Framework XML Schemas.
Server	An application that assists vendors in converting their messages to BizTalk format. For more information, visit **`www.microsoft.com/biztalkserver`**
JumpStart Kit	A set of tools for developing BizTalk applications.

Fig. 18.26 BizTalk terminology.

```
<?xml version = "1.0"?>
<BizTalk xmlns =
   "urn:schemas-biztalk-org:BizTalk/biztalk-0.81.xml">

<!-- Fig. 18.27: ìbiztalkmarkup.xml      -->
<!-- Example of standard BizTalk markup -->

   <Route>
      <From locationID = "8888888" locationType = "DUNS"
         handle = "23" />

      <To locationID = "454545445" locationType = "DUNS"
         handle = "45" />
   </Route>

```

Fig. 18.27 BizTalk markup using an offer Schema. (Part 1 of 3.)

```
      <Body>
         <Offers xmlns =
            "x-schema:http://schemas.biztalk.org/eshop_msn_com/
t7ntoqnq.xml">
            <Offer>
               <Model>12-a-3411d</Model>
               <Manufacturer>ExComp, Inc.</Manufacturer>
               <ManufacturerModel>DCS-48403</ManufacturerModel>

               <MerchantCategory>
                  Clothes | Sports wear
               </MerchantCategory>

               <MSNClassId></MSNClassId>

               <StartDate>2001-06-05 T13:12:00</StartDate>
               <EndDate>2001-12-05T13:12:00</EndDate>

               <RegularPrice>89.99</RegularPrice>
               <CurrentPrice>25.99</CurrentPrice>
               <DisplayPrice value = "3" />
               <InStock value = "15" />

               <ReferenceImageURL>
                  http://www.Example.com/clothes/index.jpg
               </ReferenceImageURL>

               <OfferName>Clearance sale</OfferName>

               <OfferDescription>
                  This is a clearance sale
               </OfferDescription>

               <PromotionalText>Free Shipping</PromotionalText>

               <Comments>
                  Clothes that you would love to wear.
               </Comments>

               <IconType value = "BuyNow" />

               <ActionURL>
                  http://www.example.com/action.htm
               </ActionURL>

               <AgeGroup1 value = "Infant" />
               <AgeGroup2 value = "Adult" />

               <Occasion1 value = "Birthday" />
               <Occasion2 value = "Anniversary" />
               <Occasion3 value = "Christmas" />

            </Offer>
```

Fig. 18.27 BizTalk markup using an offer Schema. (Part 2 of 3.)

```
68         </Offers>
69      </Body>
70   </BizTalk>
```

Fig. 18.27 BizTalk markup using an offer Schema. (Part 3 of 3.)

All Biztalk documents have the root element `BizTalk` (line 2). Line 3 defines a default namespace for the `BizTalk` framework elements. Element `Route` (lines 8–14) contains the routing information, which is mandatory for all BizTalk documents. Element `Route` also contains elements `To` and `From` (lines 9–12), which indicate the document's destination and source, respectively. This makes it easier for the receiving application to communicate with the sender. Attribute `locationType` specifies the type of business that sends or receives the information, and attribute `locationID` specifies a business identity (the unique identifier for a business). These attributes facilitate source and destination organization. Attribute `handle` provides information to routing applications that handle the document.

Element `Body` (lines 16–69) contains the actual message, whose Schema is defined by the businesses themselves. Lines 17–18 specify the default namespace for element `Offers` (lines 17–68), which is contained in element `Body` (note that line 18 wraps—if we split this line, Internet Explorer cannot locate the namespace). Each offer is marked up using an `Offer` element (lines 19–67) that contains elements describing the offer. Note that the tags all are business-related elements, and easily understood. For additional information on BizTalk, visit `www.biztalk.com`.

In this chapter, we studied the Extensible Markup Language and several of its related technologies. In Chapter 19, we begin our discussion of databases, which are crucial to the development of multi-tier Web-based applications.

18.8 Internet and World Wide Web Resources

`www.w3.org/xml`
The W3C (World Wide Web Consortium) facilitates the development of common protocols to ensure interoperability on the Web. Their XML page includes information about upcoming events, publications, software and discussion groups. Visit this site to read about the latest developments in XML.

`www.xml.org`
`xml.org` is a reference for XML, DTDs, schemas and namespaces.

`www.w3.org/style/XSL`
This W3C page provides information on XSL, including topics such as XSL development, learning XSL, XSL-enabled tools, XSL specification, FAQs and XSL history.

`www.w3.org/TR`
This is the W3C technical reports and publications page. It contains links to working drafts, proposed recommendations and other resources.

`www.xmlbooks.com`
This site provides a list of XML books recommended by Charles Goldfarb, one of the original designers of GML (General Markup Language), from which SGML was derived.

`www.xml-zone.com`
The Development Exchange XML Zone is a complete resource for XML information. This site includes a FAQ, news, articles and links to other XML sites and newsgroups.

`wdvl.internet.com/Authoring/Languages/XML`
Web Developer's Virtual Library XML site includes tutorials, a FAQ, the latest news and extensive links to XML sites and software downloads.

`www.xml.com`
`XML.com` provides the latest news and information about XML, conference listings, links to XML Web resources organized by topic, tools and other resources.

`msdn.microsoft.com/xml/default.asp`
The MSDN Online XML Development Center features articles on XML, Ask the Experts chat sessions, samples and demos, newsgroups and other helpful information.

`msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp`
The microsoft XML validator, which can be downloaded from this site, can validate both online and offline documents.

`www.oasis-open.org/cover/xml.html`
The SGML/XML Web Page is an extensive resource that includes links to several FAQs, online resources, industry initiatives, demos, conferences and tutorials.

`www.gca.org/whats_xml/default.htm`
The GCA site offers an XML glossary, list of books, brief descriptions of the draft standards for XML and links to online drafts.

`www-106.ibm.com/developerworks/xml`
The IBM XML Zone site is a great resource for developers. It provides news, tools, a library, case studies and information about events and standards.

`developer.netscape.com/tech/xml/index.html`
The XML and Metadata Developer Central site has demos, technical notes and news articles related to XML.

`www.projectcool.com/developer/xmlz`
The Project Cool Developer Zone site includes several tutorials covering introductory through advanced XML topics.

`www.ucc.ie/xml`
This site is a detailed XML FAQ. Developers can check out responses to some popular questions, or submit their own questions through the site.

SUMMARY

- XML is a widely supported, open technology (i.e., non-proprietary technology) for data exchange. XML is quickly becoming the standard by which applications maintain data.
- XML is highly portable. Any text editor that supports ASCII or Unicode characters can render or display XML documents. Because XML elements describe the data they contain, they are both human and machine readable.
- XML permits document authors to create custom markup for virtually any type of information. This extensibility enables document authors to create entirely new markup languages that describe specific types of data, including mathematical formulas, chemical molecular structures, music, recipes, etc.
- The processing of XML documents—which programs typically store in files whose names end with the **`.xml`** extension—requires a program called an XML parser. A parser is responsible for identifying components of XML documents then for storing those components in a data structure for manipulation.

- An XML document can reference another optional document that defines the XML document's structure. Two types of optional structure-defining documents are Document Type Definitions (DTDs) and Schemas.
- An XML document begins with an optional XML declaration, which identifies the document as an XML document. The **`version`** information parameter specifies the version of XML syntax that is used in the document.
- XML comments begin with **`<!--`** and end with **`-->`**. Data is marked up with tags whose names are enclosed in angle brackets (**`<>`**). Tags are used in pairs to delimit markup. A tag that begins markup is called a start tag, and a tag that terminates markup is called an end tag. End tags differ from start tags in that they contain a forward slash (**`/`**) character.
- Individual units of markup are called elements, which are the most fundamental XML building blocks. XML documents contain one element, called a root element, that contains every other element in the document. Elements are embedded or nested within each other to form hierarchies, with the root element at the top of the hierarchy.
- XML element names can be of any length and can contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore.
- When a user loads an XML document into Internet Explorer (IE), msxml parses the document and passes the parsed data to IE. IE then uses a style sheet to format the data.
- IE displays minus (**–**) and plus (**+**) signs next to all container elements (i.e., elements that contain other elements). A minus sign indicates that all child elements (i.e., nested elements) are being displayed. When clicked, a minus sign becomes a plus sign (which collapses the container element and hides all children), and vice versa.
- In addition to being placed between tags, data also can be placed in attributes, which are name–value pairs in start tags. Elements can have any number of attributes.
- Because XML allows document authors to create their own tags, naming collisions (i.e., two different elements that have the same name) can occur. As in C#, XML namespaces provide a means for document authors to prevent collisions. Namespace prefixes are prepended to elements to specify the namespace to which the element belongs.
- Each namespace prefix is bound to a uniform resource identifier (URI) that uniquely identifies the namespace. A URI is a series of characters that differentiate names. Document authors create their own namespace prefixes. Virtually any name can be used as a namespace prefix except the reserved namespace prefix **`xml`**.
- To eliminate the need to place a namespace prefix in each element, document authors can specify a default namespace for an element and its children.
- When an XML parser successfully parses a document, the parser stores a tree structure containing the document's data in memory. This hierarchical tree structure is called a Document Object Model (DOM) tree. The DOM tree represents each component of the XML document as a node in the tree. Nodes that contain other nodes (called child nodes) are called parent nodes. Nodes that have the same parent are called sibling nodes. A node's descendant nodes include that node's children, its children's children and so on. A node's ancestor nodes include that node's parent, its parent's parent and so on. The DOM tree has a single root node that contains all other nodes in the document.
- Namespace **`System.Xml`**, contains classes for creating, reading and manipulating XML documents.
- **`XmlReader`**-derived class **`XmlNodeReader`** iterates through each node in the XML document.
- Class **`XmlReader`** is an **`abstract`** class that defines the interface for reading XML documents.

- An **`XmlDocument`** object conceptually represents an empty XML document.
- The XML documents are parsed and loaded into an **`XmlDocument`** object when method **`Load`** is invoked. Once an XML document is loaded into an **`XmlDocument`**, its data can be read and manipulated programmatically.
- An **`XmlNodeReader`** allows us to read one node at a time from an **`XmlDocument`**.
- Method **`Read`** of **`XmlReader`** reads one node from the DOM tree.
- The **`Name`** property contains the node's name, the **`Value`** property contains the node's data and the **`NodeType`** property contains the node type (i.e., element, comment, text etc.).
- Line breaks use the character sequence **`"\r\n"`**, which denotes a carriage return followed by a line feed. This is the standard line break for Windows-based applications and controls.
- Method **`CreateNode`** of **`XmlDocument`** takes a **`NodeType`**, a **`Name`** and a **`NamespaceURI`** as arguments.
- An **`XmlTextWriter`** streams XML data to disk. Method **`WriteTo`** writes an XML representation to an **`XmlTextWriter`** stream.
- An **`XmlTextReader`** reads XML data from a file.
- Class **`XPathNavigator`** in the **`System.Xml.XPath`** namespace can iterate through node lists that match search criteria, written as an XPath expression.
- XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies.
- Navigation methods of **`XPathNavigator`** are **`MoveToFirstChild`**, **`MoveToParent`**, **`MoveToNext`** and **`MoveToPrevious`**. Each method performs the action that its name implies: Method **`MoveToFirstChild`** moves to the first child of the node referenced by the **`XPathNavigator`**, **`MoveToParent`** moves to the parent node of the node referenced by the **`XPathNavigator`**, **`MoveToNext`** moves to the next sibling of the node referenced by the **`XPathNavigator`** and **`MoveToPrevious`** moves to the previous sibling of the node referenced by the **`XPathNavigator`**.
- Whereas XML contains only data, XSLT is capable of converting XML into any text based document. XSLT documents typically have the extension **`.xsl`**.
- When transforming an XML document via XSLT, two tree structures are involved: The source tree, which is the XML document being transformed, and the result tree, which is the result (e.g., XHTML) of the transformation.
- XSLT specifies the use of element **`value-of`** to retrieve an attribute's value. The symbol **`@`** specifies an attribute node.
- The node-set function **`name`** retrieves the current node's element name.
- Attribute **`select`** selects the value of context node's attribute.
- XML documents can be transformed programmatically through C#. The **`System.Xml.Xsl`** namespace facilities the application of XSLT style sheets to XML documents.
- Class **`XsltArgumentList`** is a list of **`string`** parameters that can be applied to a style sheet.
- BizTalk consists of three parts: The BizTalk Server, the BizTalk Framework and the BizTalk Schema Library.
- The BizTalk Server (BTS) parses and translates all inbound and outbound messages (or documents) going to and from a business.
- The BizTalk Framework is a Schema for structuring those messages.

- The BizTalk Schema Library is a collection of different Framework Schemas. Businesses can design their own Schema or choose one from the BizTalk Schema Library.
- All Biztalk documents have the root element **`BizTalk`**.

TERMINOLOGY

`@` character
`\r\n`
`Add` method
ancestor node
asterisk (**`*`**) occurrence indicator
`ATTLIST`
attribute
attribute node
attribute value
BizTalk Framework
BizTalk Schema Library
BizTalk Server (BTS)
`CDATA` character data
child element
child node
container element
context node
`CreateNavigator` method
`CreateNode` method
`Current` property
`data-type` attribute
default namespace
descendant node
`doctype-public` attribute
`doctype-system` attribute
document root
Document Type Definition (DTD)
DOM (Document Object Model)
EBNF (Extended Backus-Naur Form) grammar
`ELEMENT` element type declaration
empty element
`EMPTY` keyword
end tag
Extensible Stylesheet Language (XSL)
external DTD
forward slash
`#IMPLIED` flag
invalid document
`IsEmptyElement` property
`LastChild` property
`Load` method
`match` attribute
markup
`maxOccurs` attribute
`method` attribute
`minOccurs` attribute
`MoveToFirstChild` property
`MoveToNext` property
`MoveToParent` property
`MoveToPrevious` property
`MoveToRoot` property
msxml parser
`name` attribute
`name` node-set function
`Name` property
namespace prefix
node
`Nodes` collection
node-set function
`NodeType` property
nonvalidating XML parser
occurrence indicator
`omit-xml-declaration` attribute
`order` attribute
parent node
`Parent` property
`ParentNode` property
parsed character data
parser
`#PCDATA` flag
PI (processing instruction)
PI target
PI value
plus-sign (**`+`**) occurrence indicator
processing instruction
question-mark (**`?`**) occurrence indicator
`Read` Method
recursive descent
reserved namespace prefix **`xml`**
result tree
root element
root node
`Schema` element
`schema` property
`Schemas` property
`select` attribute
`Select` method
sibling node
single-quote character (**`'`**)
source tree

style sheet
`sum` function
`SYSTEM` flag
`System.Xml` namespace
`System.Xml.Schema` namespace
text node
`Transform` method
tree-based model
`type` attribute
validating XML parser
`ValidatingReader` class
`ValidationEventHandler` class
`ValidationType` property
`ValidationType.Auto` constant
`value` property
`version` attribute
`version` information parameter
W3C XML Schema
well-formed document
`.xdr` extension
XML (Extensible Markup Language)
XML declaration
`.xml` file extension
`xml` namespace
XML node
XML processor
XML Schema
XML Validator
`XmlDocument` class
`XmlNodeReader` class
`XmlNodeType` enumeration
`XmlNodeType.Comment` constant
`XmlNodeType.Element` constant
`XmlNodeType.EndElement` constant
`XmlNodeType.Text` constant
`XmlNodeType.XmlDeclaration` constant
`xmlns` attribute
`XmlPathNodeIterator` class
`XmlReader` class
`XmlSchema` class
`XmlSchemaCollection` collection
`XmlTextWriter` class
`XPathExpression` class
`XPathNavigator` class
`.xsl` extension
XSL Transformations (XSLT)
XSL variable
`xsl:apply-templates` element
`xsl:for-each` element
`xsl:output` element
`xsl:sort` element
`xsl:stylesheet` element
`xsl:template` element
`xsl:value-of` element
`XslTransform` class
`XsltTextWriter` class

SELF-REVIEW EXERCISES

18.1 Which of the following are valid XML element names?

a) `yearBorn`
b) `year.Born`
c) `year Born`
d) `year-Born1`
e) `2_year_born`
f) `--year/born`
g) `year*born`
h) `.year_born`
i) `_year_born_`
j) `y_e-a_r-b_o-r_n`

18.2 State whether the following are *true* or *false*. If *false*, explain why.

a) XML is a technology for creating markup languages.
b) XML markup is delimited by forward and backward slashes (`/` and `\`).
c) All XML start tags must have corresponding end tags.
d) Parsers check an XML document’s syntax.
e) XML does not support namespaces.
f) When creating new XML elements, document authors must use the set of XML tags provided by the W3C.

g) The pound character (**#**), the dollar sign (**$**), ampersand (**&**), greater-than (**>**) and less-than (**<**) are examples of XML reserved characters.

18.3 Fill in the blanks for each of the following statements:

a) __________ help prevent naming collisions.
b) __________ embed application–specific information into an XML document.
c) __________ is Microsoft's XML parser.
d) XSL element __________ writes a **`DOCTYPE`** to the result tree.
e) Microsoft XML Schema documents have root element __________.
f) To define an element attribute in a DTD, __________ is used.
g) XSL element __________ is the root element in an XSL document.
h) XSL element __________ selects specific XML elements using repetition.

18.4 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a) XML is not case sensitive.
b) C# architecture supports W3C Schema.
c) DTDs are a vocabulary of XML.
d) Schema is a technology for locating information in an XML document.

18.5 In Fig. 18.1, we subdivided the **`author`** element into more detailed pieces. How might you subdivide the **`date`** element?

18.6 Write a processing instruction that includes the stylesheet **`wap.xsl`** for use in Internet Explorer.

18.7 Fill in the blanks in each of the following statements:

a) Nodes that contain other nodes are called __________ nodes.
b) Nodes that are peers are called __________ nodes.
c) Class **`XmlDocument`** is analogous to the __________ of a tree.
d) Method __________ adds an **`XmlNode`** to an **`XmlTree`** as a child of the current node.

18.8 Write an XPath expression that locates **`contact`** nodes in **`letter.xml`** (Fig. 18.3).

18.9 Describe the **`Select`** method of **`XPathNavigator`**.

ANSWERS TO SELF-REVIEW EXERCISES

18.1 a, b, d, i, j. [Choice c is incorrect because it contains a space; Choice e is incorrect because the first character is a number; Choice f is incorrect because it contains a division symbol (**/**) and does not begin with a letter or underscore; Choice g is incorrect because it contains an asterisk (*****); Choice h is incorrect because the first character is a period (**.**) and does not begin with a letter or underscore.]

18.2 a) True. b) False. In an XML document, markup text is delimited by angle brackets (**<** and **>**), with a forward slash in the end tag. c) True. d) True. e) False. XML does support namespaces. f) False. When creating new tags, document authors can use any valid name except the reserved word **`xml`** (also **`XML`**, **`Xml`** etc.). g) False. XML reserved characters include the ampersand (**&**), the left-angle bracket (**<**) and the right-angle bracket (**>**), but not **#** and **$**.

18.3 a) namespaces. b) processing instructions. c) msxml. d) **`xsl:output`**. e) **`Schema`**. f) an operator (**`mo`**). g) **`xsl:stylesheet`**. h) **`xsl:for-each`**.

18.4 a) False. XML is case sensitive. b) True. c) False. DTDs use EBNF grammar which is not XML syntax. d) False. XPath is a technology for locating information in an XML document.

18.5
```
<date>
   <month>December</month>
   <day>6</day>
   <year>2001</year>
</date>.
```

18.6 `<?xsl:stylesheet type = "text/xsl" href = "wap.xsl"?>`

18.7 a) parent. b) sibling. c) root. e) `AppendChild`.

18.8 `/letter/contact`.

18.9 `Select` takes either an `XPathExpression` or a `string` argument containing an `XPathExpression` to select nodes referenced by the navigator.

EXERCISES

18.10 Create an XML document that marks up the nutrition facts for a package of cookies. A package of cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 2 grams of saturated fat, 5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugars and 5 grams of protein. Name this document `nutrition.xml`. Load the XML document into Internet Explorer [*Hint*: Your markup should contain elements describing the product name, serving size/amount, calories, sodium, cholesterol, proteins, etc. Mark up each nutrition fact/ingredient listed above.]

18.11 Write an XSLT style sheet for your solution to Exercise 18.10 that displays the nutritional facts in an XHTML table. Modify Fig. 18.25 (`TransformTest.cs`) to output an XHTML file, `nutrition.html`. Render `nutrition.html` in a Web browser.

18.12 Write a Microsoft Schema for Fig. 18.23.

18.13 Alter Fig. 18.20 (`ValidationTest.cs`) to include a list of Schemas in a drop-down box, along with the list of XML files. Allow the user to test for whether any XML file on the list satisfies a specific Schema. Use `books.xml`, `books.xsd`, `nutrition.xml`, `nutrition.xsd` and `fail.xml`.

18.14 Modify `XmlReaderTest` (Fig. 18.7) to display `letter.xml` (Fig. 18.3) in a `TreeView`, instead of in a text box.

18.15 Modify Fig. 18.24 (`sorting.xsl`) to sort each section (i.e., frontmatter, chapters and appendices) by page number rather than by chapter number. Save the modified document as `sorting_byChapter.xsl`.

18.16 Modify `XmlTransform.cs` (Fig. 18.25) to take in `sorting.xml` (Fig. 18.23), `sorting.xsl` (Fig. 18.24) and `sorting_byChapter.xsl`, and print the XHTML document resulting from the transform of `sorting.xml` into two XHTML files, `sorting_byPage.html` and `sorting_byChapter.html`.

19

Database, SQL and ADO .NET

Objectives

- To understand the relational database model.
- To understand basic database queries using Structured Query Language (SQL).
- To use the classes and interfaces of namespace **`System.Data`** to manipulate databases.
- To understand and use ADO .NET's disconnected model.
- To use the classes and interfaces of namespace **`System.Data.OleDb`**.

It is a capital mistake to theorize before one has data.
Arthur Conan Doyle

Now go, write it before them in a table, and note it in a book, that it may be for the time to come for ever and ever.
The Holy Bible: The Old Testament

Let's look at the record.
Alfred Emanuel Smith

Get your facts first, and then you can distort them as much as you please.
Mark Twain

I like two kinds of men: domestic and foreign.
Mae West

Outline

19.1 Introduction
19.2 Relational Database Model
19.3 Relational Database Overview: `Books` Database
19.4 Structured Query Language (SQL)
  19.4.1 Basic `SELECT` Query
  19.4.2 `WHERE` Clause
  19.4.3 `ORDER BY` Clause
  19.4.4 Merging Data from Multiple Tables: `INNER JOIN`
  19.4.5 Joining Data from Tables `Authors`, `AuthorISBN`, `Titles` and `Publishers`
  19.4.6 `INSERT` Statement
  19.4.7 `UPDATE` Statement
  19.4.8 `DELETE` Statement
19.5 ADO .NET Object Model
19.6 Programming with ADO .NET: Extracting Information from a DBMS
  19.6.1 Connecting to and Querying an Access Data Source
  19.6.2 Querying the `Books` Database
19.7 Programming with ADO.NET: Modifying a DBMS
19.8 Reading and Writing XML Files

Summary • *Terminology* • *Self-Review Exercises* • *Answers to Self-Review Exercises* • *Exercises* • *Bibliography*

19.1 Introduction

A *database* is an integrated collection of data. Many different strategies exist for organizing data in databases to facilitate easy access to and manipulation of the data. A *database management system* (*DBMS*) provides mechanisms for storing and organizing data in a manner that is consistent with the database's format. Database management systems enable programmers to access and store data without worrying about the internal representation of databases.

Today's most popular database systems are *relational databases*. Almost universally, relational databases use a language called *Structured Query Language* (*SQL*—pronounced as its individual letters or as "sequel") to perform *queries* (i.e., to request information that satisfies given criteria) and to manipulate data. [*Note*: The writing in this chapter assumes that SQL is pronounced as its individual letters. For this reason, we often precede SQL with the article "an," as in "an SQL database" or "an SQL statement."]

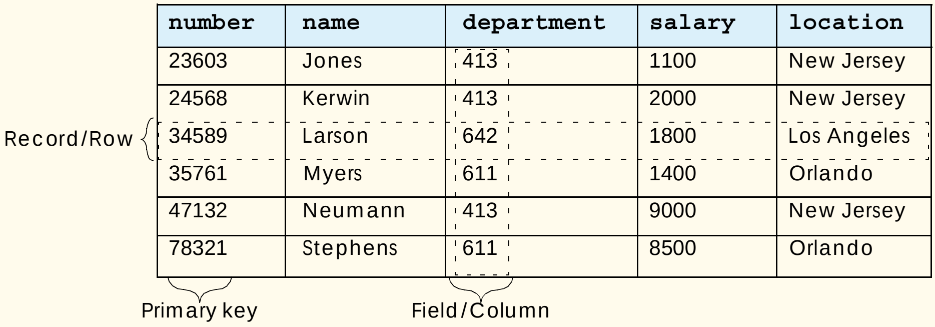
Some popular, enterprise-level relational database systems include Microsoft SQL Server, Oracle™, Sybase™, DB2™, Informix™ and MySQL™. This chapter presents examples using Microsoft Access—a relational database system that is packaged with Microsoft Office.

A programming language connects to, and interacts with, a relational database via an *interface*—software that facilitates communication between a database management system and a program. C# programmers communicate with databases and manipulate their data through *Microsoft ActiveX Data Objects*™ (ADO), *ADO .NET*.

19.2 Relational Database Model

The *relational database model* is a logical representation of data that allows relationships among data to be considered without concern for the physical structure of the data. A relational database is composed of *tables*. Figure 19.1 illustrates an example table that might be used in a personnel system. The table name is **`Employee`**, and its primary purpose is to illustrate the specific attributes of various employees. A particular row of the table is called a *record* (or *row*). This table consists of six records. The **`number`** *field* (or *column*) of each record in the table is the *primary key* for referencing data in the table. A primary key is a field (or fields) in a table that contain(s) unique data—i.e, data that is not duplicated in other records of that table. This guarantees that each record can be identified by at least one distinct value. Examples of primary-key fields are columns that contain social security numbers, employee IDs and part numbers in an inventory system. The records of Fig. 19.1 are *ordered* by primary key. In this case, the records are listed in increasing order (they also could be listed in decreasing order).

Each column of the table represents a different field. Records normally are unique (by primary key) within a table, but particular field values might be duplicated in multiple records. For example, three different records in the **`Employee`** table's **`Department`** field contain the number 413.



`number`	**`name`**	**`department`**	**`salary`**	**`location`**
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Record/Row
Primary key
Field/Column

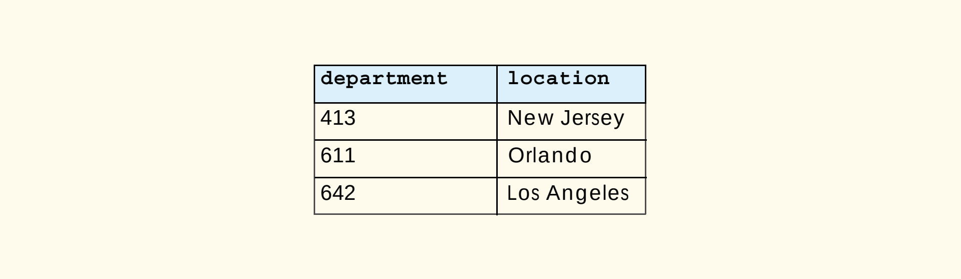
Fig. 19.1 Relational-database structure of an **`Employee`** table.

Often, different users of a database are interested in different data and different relationships among those data. Some users require only subsets of the table columns. To obtain table subsets, we use SQL statements to specify certain data we wish to *select* from a table. SQL provides a complete set of commands (including ***`SELECT`***) that enable programmers to define complex *queries* to select data from a table. The results of a query commonly are called *result sets* (or *record sets*). For example, we might select data from the table in Fig. 19.1 to create a new result set containing only the location of each department. This result set appears in Fig. 19.2. SQL queries are discussed in detail in Section 19.4.

19.3 Relational Database Overview: Books Database

The next section provides an overview of SQL in the context of a sample **`Books`** database that we created for this chapter. However, before we discuss SQL, we must explain the various tables of the **`Books`** database. We use this database to introduce various database concepts, including the use of SQL to manipulate and obtain useful information from the database. We provide a script to create the database, which is located in the Chapter 19 examples directory on the CD accompanying this book. Section 19.6 explains how to use the script.

The database consists of four tables: **`Authors`**, **`Publishers`**, **`AuthorISBN`** and **`Titles`**. The **`Authors`** table (described in Fig. 19.3) consists of three fields (or columns) that maintain each author's unique ID number, first name and last name. Figure 19.4 contains the data from the **`Authors`** table of the **`Books`** database.



department	location
413	New Jersey
611	Orlando
642	Los Angeles

Fig. 19.2 Result set formed by selecting **`Department`** and **`Location`** data from the **`Employee`** table.

Field	Description
`authorID`	Author's ID number in the database. In the **`Books`** database, this **`int`** field is defined as an *auto-incremented field*. For each new record inserted in this table, the database increments the **`authorID`** value, ensuring that each record has a unique **`authorID`**. This field represents the table's primary key.
`firstName`	Author's first name (a **`string`**).
`lastName`	Author's last name (a **`string`**).

Fig. 19.3 **`Authors`** table from **`Books`**.

The **`Publishers`** table (described in Fig. 19.5) consists of two fields, representing each publisher's unique ID and name. Figure 19.6 contains the data from the **`Publishers`** table of the **`Books`** database.

The **`AuthorISBN`** table (described in Fig. 19.7) consists of two fields that maintain the authors' ID numbers and the corresponding ISBN numbers of their books. This table helps associate the names of the authors with the titles of their books. Figure 19.8 contains the data from the **`AuthorISBN`** table of the **`Books`** database. ISBN is an abbreviation for "International Standard Book Number"—a numbering scheme by which publishers worldwide assign every book a unique identification number. [*Note*: To save space, we have split the contents of this figure into two columns, each containing the **`authorID`** and **`isbn`** fields.

`authorID`	**`firstName`**	**`lastName`**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

Fig. 19.4 Data from the **`Authors`** table of **`Books`**.

Field	**Description**
`publisherID`	The publisher's ID number in the database. This auto-incremented **`int`** field is the table's primary-key field.
`publisherName`	The name of the publisher (a **`string`**).

Fig. 19.5 **`Publishers`** table from **`Books`**.

`publisherID`	`publisherName`
1	Prentice Hall
2	Prentice Hall PTG

Fig. 19.6 Data from the **`Publishers`** table of **`Books`**.

Field	Description
`authorID`	The author's ID number, which allows the database to associate each book with a specific author. The integer ID number in this field must also appear in the **`Authors`** table.
`isbn`	The ISBN number for a book (a **`string`**).

Fig. 19.7 **`AuthorISBN`** table from **`Books`**.

`authorID`	`isbn`	`authorID`	`isbn`
1	0130895725	2	0139163050
1	0132261197	2	013028419x
1	0130895717	2	0130161438
1	0135289106	2	0130856118
1	0139163050	2	0130125075
1	013028419x	2	0138993947
1	0130161438	2	0130852473
1	0130856118	2	0130829277
1	0130125075	2	0134569555
1	0138993947	2	0130829293
1	0130852473	2	0130284173
1	0130829277	2	0130284181
1	0134569555	2	0130895601
1	0130829293	3	013028419x
1	0130284173	3	0130161438
1	0130284181	3	0130856118
1	0130895601	3	0134569555
2	0130895725	3	0130829293
2	0132261197	3	0130284173
2	0130895717	3	0130284181
2	0135289106	4	0130895601

Fig. 19.8 Data from **`AuthorISBN`** table in **`Books`**.

The **`Titles`** table (described in Fig. 19.9) consists of seven fields that maintain general information about the books in the database. This information includes each book's ISBN number, title, edition number, copyright year and publisher's ID number, as well as the name of a file containing an image of the book cover and, finally, each book's price. Figure 19.10 contains the data from the **`Titles`** table.

Field	Description
`isbn`	ISBN number of the book (a **`string`**).
`title`	Title of the book (a **`string`**).
`editionNumber`	Edition number of the book (a **`string`**).
`copyright`	Copyright year of the book (an **`int`**).
`publisherID`	Publisher's ID number (an **`int`**). This value must correspond to an ID number in the **`Publishers`** table.
`imageFile`	Name of the file containing the book's cover image (a **`string`**).
`price`	Suggested retail price of the book (a real number). [*Note*: The prices shown in this database are for example purposes only.]

Fig. 19.9 **`Titles`** table from **`Books`**.

`isbn`	**`title`**	**`editionNumber`**	**`publisherID`**	**`copyright`**	**`imageFile`**	**`price`**
0130923613	Python How to Program	1	1	2002	**`python.jpg`**	$69.95
0130622214	C# How to Program	1	1	2002	**`cshtp.jpg`**	$69.95
0130341517	Java How to Program	4	1	2002	**`jhtp4.jpg`**	$69.95
0130649341	The Complete Java Training Course	4	2	2002	**`javactc4.jpg`**	$109.95
0130895601	Advanced Java 2 Platform How to Program	1	1	2002	**`advjhtp1.jpg`**	$69.95
0130308978	Internet and World Wide Web How to Program	2	1	2002	**`iw3htp2.jpg`**	$69.95
0130293636	Visual Basic .NET How to Program	2	1	2002	**`vbnet.jpg`**	$69.95
0130895636	The Complete C++ Training Course	3	2	2001	**`cppctc3.jpg`**	$109.95

Fig. 19.10 Data from the **`Titles`** table of **`Books`**. (Part 1 of 3.)

isbn	title	editionNumber	publisherID	copyright	imageFile	price
0130895512	The Complete e-Business & e-Commerce Programming Training Course	1	2	2001	**ebecctc.jpg**	$109.95
013089561X	The Complete Internet & World Wide Web Programming Training Course	2	2	2001	**iw3ctc2.jpg**	$109.95
0130895547	The Complete Perl Training Course	1	2	2001	**perl.jpg**	$109.95
0130895563	The Complete XML Programming Training Course	1	2	2001	**xmlctc.jpg**	$109.95
0130895725	C How to Program	3	1	2001	**chtp3.jpg**	$69.95
0130895717	C++ How to Program	3	1	2001	**cpphtp3.jpg**	$69.95
013028419X	e-Business and e-Commerce How to Program	1	1	2001	**ebechtp1.jpg**	$69.95
0130622265	Wireless Internet and Mobile Business How to Program	1	1	2001	**wireless.jpg**	$69.95
0130284181	Perl How to Program	1	1	2001	**perlhtp1.jpg**	$69.95
0130284173	XML How to Program	1	1	2001	**xmlhtp1.jpg**	$69.95
0130856118	The Complete Internet and World Wide Web Programming Training Course	1	2	2000	**iw3ctc1.jpg**	$109.95
0130125075	Java How to Program (Java 2)	3	1	2000	**jhtp3.jpg**	$69.95
0130852481	The Complete Java 2 Training Course	3	2	2000	**javactc3.jpg**	$109.95
0130323640	e-Business and e-Commerce for Managers	1	1	2000	**ebecm.jpg**	$69.95
0130161438	Internet and World Wide Web How to Program	1	1	2000	**iw3htp1.jpg**	$69.95

Fig. 19.10 Data from the **Titles** table of **Books**. (Part 2 of 3.)

isbn	title	editionNumber	publisherID	copyright	imageFile	price
0130132497	Getting Started with Visual C++ 6 with an Introduction to MFC	1	1	1999	`gsvc.jpg`	$49.95
0130829293	The Complete Visual Basic 6 Training Course	1	2	1999	`vbctc1.jpg`	$109.95
0134569555	Visual Basic 6 How to Program	1	1	1999	`vbhtp1.jpg`	$69.95
0132719746	Java Multimedia Cyber Classroom	1	2	1998	`javactc.jpg`	$109.95
0136325890	Java How to Program	1	1	1998	`jhtp1.jpg`	$69.95
0139163050	The Complete C++ Training Course	2	2	1998	`cppctc2.jpg`	$109.95
0135289106	C++ How to Program	2	1	1998	`cpphtp2.jpg`	$49.95
0137905696	The Complete Java Training Course	2	2	1998	`javactc2.jpg`	$109.95
0130829277	The Complete Java Training Course (Java 1.1)	2	2	1998	`javactc2.jpg`	$99.95
0138993947	Java How to Program (Java 1.1)	2	1	1998	`jhtp2.jpg`	$49.95
0131173340	C++ How to Program	1	1	1994	`cpphtp1.jpg`	$69.95
0132261197	C How to Program	2	1	1994	`chtp2.jpg`	$49.95
0131180436	C How to Program	1	1	1992	`chtp.jpg`	$69.95

Fig. 19.10 Data from the `Titles` table of `Books`. (Part 3 of 3.)

Figure 19.11 illustrates the relationships among the tables in the `Books` database. The first line in each table is the table's name. The field whose name appears in italics contains that table's primary key. A table's primary key uniquely identifies each record in the table. Every record must have a value in the primary-key field, and the value must be unique. This is known as the *Rule of Entity Integrity*. Note that the `AuthorISBN` table contains two fields whose names are italicized. This indicates that these two fields form a *compound primary key*—each record in the table must have a unique `authorID`–`isbn` combination. For example, several records might have an `authorID` of `2`, and several records might have an `isbn` of `0130895601`, but only one record can have both an `authorID` of `2` and an `isbn` of `0130895601`.

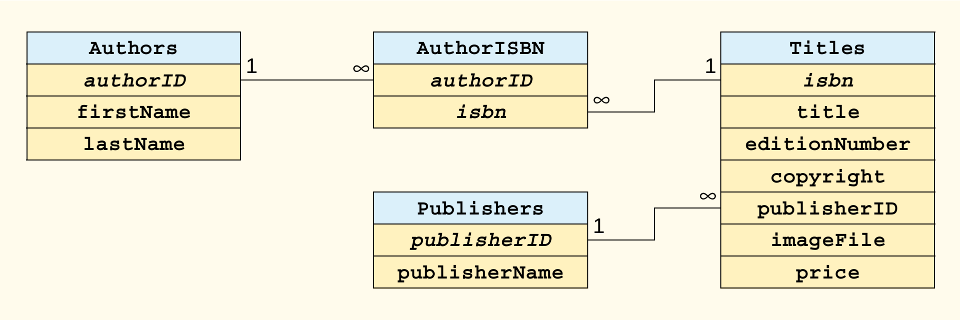


Fig. 19.11 Table relationships in **Books**.

Common Programming Error 19.1

Failure to provide a value for a primary-key field in every record breaks the Rule of Entity Integrity and causes the DBMS to report an error.

Common Programming Error 19.2

Providing duplicate values for the primary-key field of multiple records causes the DBMS to report an error.

The lines connecting the tables in Fig. 19.11 represent the *relationships* among the tables. Consider the line between the **Publishers** and **Titles** tables. On the **Publishers** end of the line, there is a **1**, and, on the **Titles** end, there is an infinity (∞) symbol. This line indicates a *one-to-many relationship*, in which every publisher in the **Publishers** table can have an arbitrarily large number of books in the **Titles** table. Note that the relationship line links the **publisherID** field in the **Publishers** table to the **publisherID** field in **Titles** table. In the **Titles** table, the **publisherID** field is a *foreign key*—a field for which every entry has a unique value in another table and where the field in the other table is the primary key for that table (e.g., **publisherID** in the **Publishers** table). Programmers specify foreign keys when creating a table. The foreign key helps maintain the *Rule of Referential Integrity*: Every foreign-key field value must appear in another table's primary-key field. Foreign keys enable information from multiple tables to be *joined* together for analysis purposes. There is a one-to-many relationship between a primary key and its corresponding foreign key. This means that a foreign-key field value can appear many times in its own table, but must appear exactly once as the primary key of another table. The line between the tables represents the link between the foreign key in one table and the primary key in another table.

Common Programming Error 19.3

Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity and causes the DBMS to report an error.

The line between the **AuthorISBN** and **Authors** tables indicates that, for each author in the **Authors** table, the **AuthorISBN** table can contain an arbitrary number of ISBNs for books written by that author. The **authorID** field in the **AuthorISBN** table is a foreign key of the **authorID** field (the primary key) of the **Authors** table. Note, again, that the line between the tables links the foreign key in table **AuthorISBN** to the corresponding primary key in table **`Authors`**. The **`AuthorISBN`** table links information in the **`Titles`** and **`Authors`** tables.

The line between the **`Titles`** and **`AuthorISBN`** tables illustrates another one-to-many relationship; a title can be written by any number of authors. In fact, the sole purpose of the **`AuthorISBN`** table is to represent a many-to-many relationship between the **`Authors`** and **`Titles`** tables; an author can write any number of books, and a book can have any number of authors.

19.4 Structured Query Language (SQL)

In this section, we provide an overview of Structured Query Language (SQL) in the context of our **`Books`** sample database. The SQL queries discussed here form the foundation for the SQL used in the chapter examples.

Figure 19.12 lists SQL keywords and provides a description of each. In the next several subsections, we discuss these SQL keywords in the context of complete SQL queries. Other SQL keywords exist, but are beyond the scope of this text. [*Note*: To locate additional information on SQL, please refer to the bibliography at the end of this chapter.]

19.4.1 Basic SELECT Query

Let us consider several SQL queries that extract information from database **`Books`**. A typical SQL query "selects" information from one or more tables in a database. Such selections are performed by ***`SELECT`* queries**. The basic format for a **`SELECT`** query is:

```
SELECT * FROM tableName
```

In this query, the asterisk (**`*`**) indicates that all columns from the *tableName* table of the database should be selected. For example, to select the entire contents of the **`Authors`** table (i.e., all data depicted in Fig. 19.4), use the query:

```
SELECT * FROM Authors
```

SQL keyword	Description
`SELECT`	Selects (retrieves) fields from one or more tables.
`FROM`	Specifies tables from which to get fields or delete records. Required in every **`SELECT`** and **`DELETE`** statement.
`WHERE`	Specifies criteria that determine the rows to be retrieved.
`INNER JOIN`	Joins records from multiple tables to produce a single set of records.
`GROUP BY`	Specifies criteria for grouping records.
`ORDER BY`	Specifies criteria for ordering records.
`INSERT`	Inserts data into a specified table.
`UPDATE`	Updates data in a specified table.
`DELETE`	Deletes data from a specified table.

Fig. 19.12 SQL query keywords.

To select specific fields from a table, replace the asterisk (`*`) with a comma-separated list of the field names to select. For example, to select only the fields **`authorID`** and **`lastName`** for all rows in the **`Authors`** table, use the query:

```
SELECT authorID, lastName FROM Authors
```

This query returns only the data presented in Fig. 19.13. [*Note*: If a field name contains spaces, the entire field name must be enclosed in square brackets (`[]`) in the query. For example, if the field name is **`first name`**, it must appear in the query as **`[first name]`**.

Common Programming Error 19.4

If a program assumes that an SQL statement using the asterisk (`*`) to select fields always returns those fields in the same order, the program could process the result set incorrectly. If the field order in the database table(s) changes, the order of the fields in the result set would change accordingly.

Performance Tip 19.1

If a program does not know the order of fields in a result set, the program must process the fields by name. This could require a linear search of the field names in the result set. If users specify the field names that they wish to select from a table (or several tables), the application receiving the result set knows the order of the fields in advance. When this occurs, the program can process the data more efficiently, because fields can be accessed directly by column number.

19.4.2 WHERE Clause

In most cases, users search a database for records that satisfy certain *selection criteria*. Only records that match the selection criteria are selected. SQL uses the optional ***`WHERE`*** *clause* in a **`SELECT`** query to specify the selection criteria for the query. The simplest format for a **`SELECT`** query that includes selection criteria is:

```
SELECT fieldName1, fieldName2, … FROM tableName WHERE criteria
```

For example, to select the **`title`**, **`editionNumber`** and **`copyright`** fields from those rows of table **`Titles`** in which the **`copyright`** date is greater than **`1999`**, use the query:]

authorID	lastName	authorID	lastName
1	Deitel	8	McPhie
2	Deitel	9	Yaeger
3	Nieto	10	Zlatkina
4	Steinbuhler	12	Wiedermann
5	Santry	12	Liperi
6	Lin	13	Listfield
7	Sadhu		

Fig. 19.13 **`authorID`** and **`lastName`** from the **`Authors`** table.

```
SELECT title, editionNumber, copyright
FROM Titles
WHERE copyright > 1999
```

Figure 19.14 shows the result set of the preceding query. [*Note*: When we construct a query for use in C#, we simply create a **`string`** containing the entire query. However, when we display queries in the text, we often use multiple lines and indentation to enhance readability.]

Performance Tip 19.2

Using selection criteria improves performance, because queries that involve such criteria normally select a portion of the database that is smaller than the entire database. Working with a smaller portion of the data is more efficient than working with the entire set of data stored in the database.

`Title`	`editionNumber`	`copyright`
Internet and World Wide Web How to Program	2	2002
Java How to Program	4	2002
The Complete Java Training Course	4	2002
The Complete e-Business & e-Commerce Programming Training Course	1	2001
The Complete Internet & World Wide Web Programming Training Course	2	2001
The Complete Perl Training Course	1	2001
The Complete XML Programming Training Course	1	2001
C How to Program	3	2001
C++ How to Program	3	2001
The Complete C++ Training Course	3	2001
e-Business and e-Commerce How to Program	1	2001
Internet and World Wide Web How to Program	1	2000
The Complete Internet and World Wide Web Programming Training Course	1	2000
Java How to Program (Java 2)	3	2000
The Complete Java 2 Training Course	3	2000
XML How to Program	1	2001
Perl How to Program	1	2001
Advanced Java 2 Platform How to Program	1	2002
e-Business and e-Commerce for Managers	1	2000
Wireless Internet and Mobile Business How to Program	1	2001
C# How To Program	1	2002
Python How to Program	1	2002
Visual Basic .NET How to Program	2	2002

Fig. 19.14 Titles with copyrights after 1999 from table **`Titles`**.

The `WHERE` clause condition can contain operators `<`, `>`, `<=`, `>=`, `=`, `<>` and `LIKE`. Operator `LIKE` is used for *pattern matching* with wildcard characters *asterisk* (`*`) and *question mark* (`?`). Pattern matching allows SQL to search for strings that “match a pattern.”

A pattern that contains an asterisk (`*`) searches for strings in which zero or more characters take the asterisk character’s place in the pattern. For example, the following query locates the records of all authors whose last names start with the letter `D`:

```
SELECT authorID, firstName, lastName
FROM Authors
WHERE lastName LIKE 'D*'
```

The preceding query selects the two records shown in Fig. 19.15, because two of the authors in our database have last names that begin with the letter `D` (followed by zero or more characters). The `*` in the `WHERE` clause’s `LIKE` pattern indicates that any number of characters can appear after the letter `D` in the `lastName` field. Notice that the pattern string is surrounded by single-quote characters.

Portability Tip 19.1

Not all database systems support the `LIKE` operator, so be sure to read the database system’s documentation carefully before employing this operator.

Portability Tip 19.2

Most databases use the `%` character in place of the `*` character in `LIKE` expressions.

Portability Tip 19.3

In some databases, string data is case sensitive.

Portability Tip 19.4

In some databases, table names and field names are case sensitive.

Good Programming Practice 19.1

By convention, SQL keywords should be written entirely in uppercase letters on systems that are not case sensitive. This emphasizes the SQL keywords in an SQL statement.

A pattern string including a question mark (`?`) character searches for strings in which exactly one character takes the question mark’s place in the pattern. For example, the following query locates the records of all authors whose last names start with any character (specified with `?`), followed by the letter `i`, followed by any number of additional characters (specified with `*`):

`authorID`	`firstName`	`lastName`
1	Harvey	Deitel
2	Paul	Deitel

Fig. 19.15 Authors from the `Authors` table whose last names start with `D`.

```
SELECT authorID, firstName, lastName
FROM Authors
WHERE lastName LIKE '?i*'
```

The preceding query produces the records listed in Fig. 19.16; five authors in our database have last names in which the letter **`i`** is the second letter.

Portability Tip 19.5

*Most databases use the _ character in place of the **`?`** character in **`LIKE`** expressions.*

19.4.3 `ORDER BY` Clause

The results of a query can be arranged in ascending or descending order using the optional ***`ORDER BY`*** *clause*. The simplest forms for an **`ORDER BY`** clause are:

```
SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field ASC
SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field DESC
```

where **`ASC`** specifies ascending order (lowest to highest), **`DESC`** specifies descending order (highest to lowest) and *field* specifies the field whose values determine the sorting order.

For example, to obtain a list of authors arranged in ascending order by last name (Fig. 19.17), use the query:

```
SELECT authorID, firstName, lastName
FROM Authors
ORDER BY lastName ASC
```

Note that the default sorting order is ascending; therefore, **`ASC`** is optional.

`authorID`	**`firstName`**	**`lastName`**
3	Tem	Nieto
6	Ted	Lin
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

Fig. 19.16 Authors from table **`Authors`** whose last names contain **`i`** as the second letter.

`authorID`	**`firstName`**	**`lastName`**
2	Paul	Deitel
1	Harvey	Deitel

Fig. 19.17 Authors from table **`Authors`** in ascending order by **`lastName`**. (Part 1 of 2.)

authorID	**firstName**	**lastName**
6	Ted	Lin
12	Jonathan	Liperi
13	Jeffrey	Listfield
8	David	McPhie
3	Tem	Nieto
7	Praveen	Sadhu
5	Sean	Santry
4	Kate	Steinbuhler
11	Ben	Wiedermann
9	Cheryl	Yaeger
10	Marina	Zlatkina

Fig. 19.17 Authors from table **`Authors`** in ascending order by **`lastName`**. (Part 2 of 2.)

To obtain the same list of authors arranged in descending order by last name (Fig. 19.18), use the query:

```
SELECT authorID, firstName, lastName
FROM Authors
ORDER BY lastName DESC
```

authorID	**firstName**	**lastName**
10	Marina	Zlatkina
9	Cheryl	Yaeger
11	Ben	Wiedermann
4	Kate	Steinbuhler
5	Sean	Santry
7	Praveen	Sadhu
3	Tem	Nieto
8	David	McPhie
13	Jeffrey	Listfield
12	Jonathan	Liperi
6	Ted	Lin
2	Paul	Deitel
1	Harvey	Deitel

Fig. 19.18 Authors from table **`Authors`** in descending order by **`lastName`**.

The **ORDER BY** clause also can be used to order records by multiple fields. Such queries are written in the form:

```
ORDER BY field1 sortingOrder, field2 sortingOrder, …
```

where *sortingOrder* is either **ASC** or **DESC**. Note that the *sortingOrder* does not have to be identical for each field.

For example, the query:

```
SELECT authorID, firstName, lastName
FROM Authors
ORDER BY lastName, firstName
```

sorts all authors in ascending order by last name, then by first name. This means that, if any authors have the same last name, their records are returned sorted by first name (Fig. 19.19).

The **WHERE** and **ORDER BY** clauses can be combined in one query. For example, the query:

```
SELECT isbn, title, editionNumber, copyright, price
FROM Titles
WHERE title
LIKE '*How to Program' ORDER BY title ASC
```

returns the ISBN, title, edition number, copyright and price of each book in the **Titles** table that has a **title** ending with "**How to Program**"; it lists these records in ascending order by **title**. The results of the query are depicted in Fig. 19.20.

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel
6	Ted	Lin
12	Jonathan	Liperi
13	Jeffrey	Listfield
8	David	McPhie
3	Tem	Nieto
7	Praveen	Sadhu
5	Sean	Santry
4	Kate	Steinbuhler
11	Ben	Wiedermann
9	Cheryl	Yaeger
10	Marina	Zlatkina

Fig. 19.19 Authors from table **Authors** in ascending order by **lastName** and by **firstName**.

isbn	**title**	**edition-Number**	**copy-right**	**price**
0130895601	Advanced Java 2 Platform How to Program	1	2002	$69.95
0131180436	C How to Program	1	1992	$69.95
0130895725	C How to Program	3	2001	$69.95
0132261197	C How to Program	2	1994	$49.95
0130622214	C# How To Program	1	2002	$69.95
0135289106	C++ How to Program	2	1998	$49.95
0131173340	C++ How to Program	1	1994	$69.95
0130895717	C++ How to Program	3	2001	$69.95
013028419X	e-Business and e-Commerce How to Program	1	2001	$69.95
0130308978	Internet and World Wide Web How to Program	2	2002	$69.95
0130161438	Internet and World Wide Web How to Program	1	2000	$69.95
0130341517	Java How to Program	4	2002	$69.95
0136325890	Java How to Program	1	1998	$49.95
0130284181	Perl How to Program	1	2001	$69.95
0130923613	Python How to Program	1	2002	$69.95
0130293636	Visual Basic .NET How to Program	2	2002	$69.95
0134569555	Visual Basic 6 How to Program	1	1999	$69.95
0130622265	Wireless Internet and Mobile Business How to Program	1	2001	$69.95
0130284173	XML How to Program	1	2001	$69.95

Fig. 19.20 Books from table **`Titles`** whose titles end with **`How to Program`** in ascending order by **`title`**.

19.4.4 Merging Data from Multiple Tables: INNER JOIN

Database designers often split related data into separate tables to ensure that a database does not store data redundantly. For example, the **`Books`** database has tables **`Authors`** and **`Titles`**. We use an **`AuthorISBN`** table to provide "links" between authors and their corresponding titles. If we did not separate this information into individual tables, we would need to include author information with each entry in the **`Titles`** table. This would result in the database storing duplicate author information for authors who wrote multiple books.

Often, it is necessary for analysis purposes to merge data from multiple tables into a single set of data. Referred to as *joining* the tables, this is accomplished via an ***`INNER JOIN`*** operation in the **`SELECT`** query. An **`INNER JOIN`** merges records from two or more tables by testing for matching values in a field that is common to the tables. The simplest format for an **`INNER JOIN`** clause is:

```
SELECT fieldName1, fieldName2, …
FROM table1
INNER JOIN table2
   ON table1.fieldName = table2.fieldName
```

The **`ON`** part of the **`INNER JOIN`** clause specifies the fields from each table that are compared to determine which records are joined. For example, the following query produces a list of authors accompanied by the ISBN numbers for books written by each author:

```
SELECT firstName, lastName, isbn
FROM Authors
INNER JOIN AuthorISBN
   ON Authors.authorID = AuthorISBN.authorID
ORDER BY lastName, firstName
```

The query merges the **`firstName`** and **`lastName`** fields from table **`Authors`** with the **`isbn`** field from table **`AuthorISBN`**, sorting the results in ascending order by **`lastName`** and **`firstName`**. Notice the use of the syntax *tableName*`.`*fieldName* in the **`ON`** part of the **`INNER JOIN`**. This syntax (called a *fully qualified name*) specifies the fields from each table that should be compared to join the tables. The "*tableName*`.`" syntax is required if the fields have the same name in both tables. The same syntax can be used in any query to distinguish among fields in different tables that have the same name. Fully qualified names that start with the database name can be used to perform cross-database queries.

Software Engineering Observation 19.1

If an SQL statement includes fields from multiple tables that have the same name, the statement must precede those field names with their table names and the dot operator (e.g., ***`Authors.authorID`****).*

Common Programming Error 19.5

In a query, failure to provide fully qualified names for fields that have the same name in two or more tables is an error.

As always, the query can contain an **`ORDER BY`** clause. Figure 19.21 depicts the results of the preceding query, ordered by **`lastName`** and **`firstName`**. [*Note*: To save space, we split the results of the query into two columns, each containing the **`firstName`**, **`lastName`** and **`isbn`** fields.]

firstName	lastName	isbn	firstName	lastName	isbn
Harvey	Deitel	0130895601			
Harvey	Deitel	0130284181			
Harvey	Deitel	0130284173			

Fig. 19.21 Authors from table **`Authors`** and ISBN numbers of the authors' books, sorted in ascending order by **`lastName`** and **`firstName`**. (Part 1 of 2.)

firstName	**lastName**	**isbn**
Harvey	Deitel	0130829293
Harvey	Deitel	0134569555
Harvey	Deitel	0130829277
Harvey	Deitel	0130852473
Harvey	Deitel	0138993947
Harvey	Deitel	0130856118
Harvey	Deitel	0130161438
Harvey	Deitel	013028419x
Harvey	Deitel	0139163050
Harvey	Deitel	0135289106
Harvey	Deitel	0130895717
Harvey	Deitel	0132261197
Harvey	Deitel	0130895725
Harvey	Deitel	0130125075
Paul	Deitel	0130284181
Paul	Deitel	0130284173
Paul	Deitel	0130829293
Paul	Deitel	0134569555
Paul	Deitel	0130829277
Paul	Deitel	0130852473
Paul	Deitel	0138993947
Paul	Deitel	0130125075
Paul	Deitel	0130856118
Paul	Deitel	0130161438
Paul	Deitel	013028419x
Paul	Deitel	0139163050
Paul	Deitel	0130895601
Paul	Deitel	0135289106
Paul	Deitel	0130895717
Paul	Deitel	0132261197
Paul	Deitel	0130895725
Tem	Nieto	0130284181
Tem	Nieto	0130284173
Tem	Nieto	0130829293
Tem	Nieto	0134569555
Tem	Nieto	0130856118
Tem	Nieto	0130161438
Tem	Nieto	013028419x

Fig. 19.21 Authors from table `Authors` and ISBN numbers of the authors' books, sorted in ascending order by `lastName` and `firstName`. (Part 2 of 2.)

19.4.5 Joining Data from Tables `Authors`, `AuthorISBN`, `Titles` and `Publishers`

The `Books` database contains one predefined query (`TitleAuthor`), which selects as its results the title, ISBN number, author's first name, author's last name, copyright year and publisher's name for each book in the database. For books that have multiple authors, the query produces a separate composite record for each author. The `TitleAuthor` query is depicted in Fig. 19.22. Figure 19.23 contains a portion of the query results.

```
SELECT Titles.title, Titles.isbn, Authors.firstName,
      Authors.lastName, Titles.copyright,
       Publishers.publisherName
FROM
   ( Publishers INNER JOIN Titles
     ON Publishers.publisherID = Titles.publisherID )
     INNER JOIN
```

Fig. 19.22 `TitleAuthor` query of `Books` database. (Part 1 of 2.)

```
      ( Authors INNER JOIN AuthorISBN
      ON Authors.authorID = AuthorISBN.authorID )
   ON Titles.isbn = AuthorISBN.isbn
ORDER BY Titles.title
```

Fig. 19.22 **`TitleAuthor`** query of **`Books`** database. (Part 2 of 2.)

`Title`	**`isbn`**	**`firstName`**	**`lastName`**	**`copyright`**	**`publisherName`**
Advanced Java 2 Platform How to Program	0130895601	Paul	Deitel	2002	Prentice Hall
Advanced Java 2 Platform How to Program	0130895601	Harvey	Deitel	2002	Prentice Hall
Advanced Java 2 Platform How to Program	0130895601	Sean	Santry	2002	Prentice Hall
C How to Program	0131180436	Harvey	Deitel	1992	Prentice Hall
C How to Program	0131180436	Paul	Deitel	1992	Prentice Hall
C How to Program	0132261197	Harvey	Deitel	1994	Prentice Hall
C How to Program	0132261197	Paul	Deitel	1994	Prentice Hall
C How to Program	0130895725	Harvey	Deitel	2001	Prentice Hall
C How to Program	0130895725	Paul	Deitel	2001	Prentice Hall
C# How To Program	0130622214	Tem	Nieto	2002	Prentice Hall
C# How To Program	0130622214	Paul	Deitel	2002	Prentice Hall
C# How To Program	0130622214	Jeffrey	Listfield	2002	Prentice Hall
C# How To Program	0130622214	Cheryl	Yaeger	2002	Prentice Hall
C# How To Program	0130622214	Marina	Zlatkina	2002	Prentice Hall
C# How To Program	0130622214	Harvey	Deitel	2002	Prentice Hall
C++ How to Program	0130895717	Paul	Deitel	2001	Prentice Hall
C++ How to Program	0130895717	Harvey	Deitel	2001	Prentice Hall
C++ How to Program	0131173340	Paul	Deitel	1994	Prentice Hall
C++ How to Program	0131173340	Harvey	Deitel	1994	Prentice Hall
C++ How to Program	0135289106	Harvey	Deitel	1998	Prentice Hall
C++ How to Program	0135289106	Paul	Deitel	1998	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Harvey	Deitel	2000	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Kate	Steinbuhler	2000	Prentice Hall

Fig. 19.23 Portion of the result set produced by the query in Fig. 19.22.

Title	isbn	firstName	lastName	copyright	publisherName
e-Business and e-Commerce for Managers	0130323640	Paul	Deitel	2000	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Harvey	Deitel	2001	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Paul	Deitel	2001	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Tem	Nieto	2001	Prentice Hall

Fig. 19.23 Portion of the result set produced by the query in Fig. 19.22.

We added indentation to the query in Fig. 19.22 to make the query more readable. Let us now break down the query into its various parts. Lines 1–3 contain a comma-separated list of the fields that the query returns; the order of the fields from left to right specifies the fields' order in the returned table. This query selects fields `title` and `isbn` from table `Titles`, fields `firstName` and `lastName` from table `Authors`, field `copyright` from table `Titles` and field `publisherName` from table `Publishers`. For purposes of clarity, we fully qualified each field name with its table name (e.g., `Titles.isbn`).

Lines 5–10 specify the `INNER JOIN` operations used to combine information from the various tables. There are three `INNER JOIN` operations. It is important to note that, although an `INNER JOIN` is performed on two tables, either of those two tables can be the result of another query or another `INNER JOIN`. We use parentheses to nest the `INNER JOIN` operations; SQL evaluates the innermost set of parentheses first and then moves outward. We begin with the `INNER JOIN`:

```
( Publishers INNER JOIN Titles
   ON Publishers.publisherID = Titles.publisherID )
```

which joins the `Publishers` table and the `Titles` table `ON` the condition that the `publisherID` numbers in each table match. The resulting temporary table contains information about each book and its publisher.

The other nested set of parentheses contains the `INNER JOIN`:

```
( Authors INNER JOIN AuthorISBN ON
   Authors.AuthorID = AuthorISBN.AuthorID )
```

which joins the `Authors` table and the `AuthorISBN` table `ON` the condition that the `authorID` fields in each table match. Remember that the `AuthorISBN` table has multiple entries for `ISBN` numbers of books that have more than one author. The third `INNER JOIN`:

```
( Publishers INNER JOIN Titles
   ON Publishers.publisherID = Titles.publisherID )
INNER JOIN
( Authors INNER JOIN AuthorISBN
   ON Authors.authorID = AuthorISBN.authorID )
ON Titles.isbn = AuthorISBN.isbn
```

joins the two temporary tables produced by the two prior inner joins **`ON`** the condition that the **`Titles.isbn`** field for each record in the first temporary table matches the corresponding **`AuthorISBN.isbn`** field for each record in the second temporary table. The result of all these **`INNER JOIN`** operations is a temporary table from which the appropriate fields are selected to produce the results of the query.

Finally, line 11 of the query:

```
ORDER BY Titles.title
```

indicates that all the records should be sorted in ascending order (the default) by title.

19.4.6 INSERT Statement

The ***`INSERT`*** statement inserts a new record in a table. The simplest form for this statement is:

```
INSERT INTO tableName ( fieldName1, fieldName2, …, fieldNameN )
   VALUES ( value1, value2, …, valueN )
```

where *tableName* is the table in which to insert the record. The *tableName* is followed by a comma-separated list of field names in parentheses. The list of field names is followed by the SQL keyword **`VALUES`** and a comma-separated list of values in parentheses. The specified values in this list must match the field names listed after the table name in both order and type (for example, if *fieldName1* is specified as the **`firstName`** field, then *value1* should be a string in single quotes representing the first name). The **`INSERT`** statement:

```
INSERT INTO Authors ( firstName, lastName )
   VALUES ( 'Sue', 'Smith' )
```

inserts a record into the **`Authors`** table. The first comma-separated list indicates that the statement provides data for the **`firstName`** and **`lastName`** fields. The corresponding values to insert, which are contained in the second comma-separated list, are **`'Sue'`** and **`'Smith'`**. We do not specify an **`authorID`** in this example, because **`authorID`** is an auto-increment field in the database. Every new record that we add to this table is assigned a unique **`authorID`** value that is the next value in the auto-increment sequence (i.e., 1, 2, 3, etc.). In this case, **`Sue Smith`** would be assigned **`authorID`** number 14. Figure 19.24 shows the **`Authors`** table after we perform the **`INSERT`** operation.

`authorID`	**`firstName`**	**`lastName`**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu

Fig. 19.24 **`Authors`** after an **`INSERT`** operation to add a record. (Part 1 of 2.)

authorID	firstName	lastName
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield
14	Sue	Smith

Fig. 19.24 `Authors` after an `INSERT` operation to add a record. (Part 2 of 2.)

Common Programming Error 19.6

SQL statements use the single-quote (`'`) character as a delimiter for strings. To specify a string containing a single quote (such as O'Malley) in an SQL statement, the string must include two single quotes in the position where the single-quote character should appear in the string (e.g., `'O''Malley'`). The first of the two single-quote characters acts as an escape character for the second. Failure to escape single-quote characters in a string that is part of an SQL statement is an SQL syntax error.

19.4.7 UPDATE Statement

An ***`UPDATE`*** statement modifies data in a table. The simplest form for an `UPDATE` statement is:

```
UPDATE tableName
   SET fieldName1 = value1, fieldName2 = value2, …, fieldNameN = valueN
   WHERE criteria
```

where *tableName* is the table in which to update a record (or records). The *tableName* is followed by keyword ***`SET`*** and a comma-separated list of field name/value pairs written in the format, *fieldName* `=` *value*. The `WHERE` clause specifies the criteria used to determine which record(s) to update. For example, the `UPDATE` statement:

```
UPDATE Authors
   SET lastName = 'Jones'
   WHERE lastName = 'Smith' AND firstName = 'Sue'
```

updates a record in the `Authors` table. The statement indicates that `lastName` will be assigned the new value `Jones` for the record in which `lastName` currently is equal to `Smith` and `firstName` is equal to `Sue`. If we know the `authorID` in advance of the `UPDATE` operation (possibly because we searched for the record previously), the `WHERE` clause could be simplified as follows:

```
WHERE AuthorID = 14
```

Figure 19.25 depicts the `Authors` table after we perform the `UPDATE` operation.

`authorID`	`firstName`	`lastName`
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield
14	Suet	Jones

Fig. 19.25 Table **`Authors`** after an **`UPDATE`** operation to change a record.

Common Programming Error 19.7

Failure to use a ***`WHERE`*** *clause with an* ***`UPDATE`*** *statement could lead to logic errors.*

19.4.8 DELETE Statement

An SQL ***`DELETE`*** statement removes data from a table. The simplest form for a **`DELETE`** statement is:

```
DELETE FROM tableName WHERE criteria
```

where *tableName* is the table from which to delete a record (or records). The **`WHERE`** clause specifies the criteria used to determine which record(s) to delete. For example, the **`DELETE`** statement:

```
DELETE FROM Authors
   WHERE lastName = 'Jones' AND firstName = 'Sue'
```

deletes the record for **`Sue Jones`** from the **`Authors`** table.

Common Programming Error 19.8

`WHERE` *clauses can match multiple records. When deleting records from a database, be sure to define a* ***`WHERE`*** *clause that matches only the records to be deleted.*

Figure 19.26 depicts the **`Authors`** table after we perform the **`DELETE`** operation.

authorID	**firstName**	**lastName**
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

Fig. 19.26 Table **`Authors`** after a **`DELETE`** operation to remove a record.

19.5 ADO .NET Object Model

The ADO .NET object model provides an API for accessing database systems programmatically. ADO .NET was created for the .NET framework and is the next generation of *ActiveX Data Objects*™ (ADO).

Namespace ***`System.Data`*** is the root namespace for the ADO .NET API. The primary namespaces for ADO .NET, ***`System.Data.OleDb`*** and ***`System.Data.SqlClient`***, contain classes that enable programs to connect with and modify datasources. Namespace **`System.Data.OleDb`** contains classes that are designed to work with any datasource, whereas the **`System.Data.SqlClient`** namespace contains classes that are optimized to work with Microsoft SQL Server 2000 databases.

Instances of class ***`System.Data.DataSet`***, which consist of a set of **`DataTable`**s and relationships among those **`DataTable`**s, represent *caches* of data—data that a program stores temporarily in local memory. The structure of a **`DataSet`** mimics the structure of a relational database. An advantage of using class **`DataSet`** is that it is *disconnected*—the program does not need a persistent connection to the datasource to work with data in a **`DataSet`**. The program connects to the datasource only during the initial population of the **`DataSet`** and then to store any changes made in the **`DataSet`**. Hence, the program does not require any active, permanent connection to the datasource.

Instances of class ***`OleDbConnection`*** (namespace **`System.Data.OleDb`**) represent connections to a datasource. An instance of class ***`OleDbDataAdapter`*** connects to a datasource through an instance of class **`OleDbConnection`** and can populate a **`DataSet`** with data from that datasource. We discuss the details of creating and populating **`DataSet`**s later in this chapter. An instance of class ***`OleDbCommand`*** (namespace **`System.Data.OleDb`**) represents an arbitrary SQL command to be executed on a datasource. A program can use instances of class **`OleDbCommand`** to manipulate a datasource through an **`OleDbConnection`**. The programmer must close the active connection to the datasource explicitly once no further changes are to be made. Unlike **`DataSet`**s, **`OleDbCommand`** objects do not cache data in local memory.

19.6 Programming with ADO .NET: Extracting Information from a DBMS

In this section, we present two examples that introduce how to connect to a database, query the database and display the results of the query. The database used in these examples is the Microsoft Access **`Books`** database that we have discussed throughout this chapter. It can be found in the project directory for the application of Fig. 19.27. Each program must specify the location of this database on the computer's hard drive. When executing these examples, readers must update this location for each program. For example, before readers can run the application in Fig. 19.27 on their computers, they must change lines 234–247 so that the code specifies the correct location of the database file.

19.6.1 Connecting to and Querying an Access Data Source

The first example (Fig. 19.27) performs a simple query on the **`Books`** database that retrieves the entire **`Authors`** table and displays the data in a ***`DataGrid`*** (a **`System.Windows.Forms`** component class that can display a datasource in a GUI). The program illustrates the process of connecting to the database, querying the database and displaying the results in a **`DataGrid`**. The discussion following the example presents the key aspects of the program. [*Note*: We present all of Visual Studio's auto-generated code in Fig. 19.27 so that readers are aware of the code that Visual Studio generates for the example.]

This example uses an Access database. To register the **`Books`** database as a datasource, select **View > Server Explorer**. Right click the **Data Connections** node in the **Server Explorer** and then double click **<Add Connection>**. In the **Provider** tab of the window that appears, choose "**Microsoft Jet 4.0 OLE DB Provider**," which is the driver for Access databases. In the **Connection** tab, click the ellipses button (**...**) to the right of the textbox for the database name, which opens the **Select Access Database** window. Go to the appropriate folder, select the **`Books`** database and click **OK**. Now, this database is listed as a connection in the **Server Explorer**. Drag the database node onto the Windows Form. This creates an **`OleDbConnection`** to the source, which the Windows Form designer displays as **`oleDbConnection1`**.

```
1  // Fig. 19.27: TableDisplay.cs
2  // Displays data from a database table.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Data;
```

Fig. 19.27 Accessing and displaying a database's data. (Part 1 of 7.)

```

// Summary description for TableDisplay.cs.
public class TableDisplay : System.Windows.Forms.Form
{
   private System.Data.DataSet dataSet1;
   private System.Data.OleDb.OleDbDataAdapter oleDbDataAdapter1;
   private System.Windows.Forms.DataGrid dataGrid1;
   private System.Data.OleDb.OleDbCommand oleDbSelectCommand1;
   private System.Data.OleDb.OleDbCommand oleDbInsertCommand1;
   private System.Data.OleDb.OleDbCommand oleDbUpdateCommand1;
   private System.Data.OleDb.OleDbCommand oleDbDeleteCommand1;
   private System.Data.OleDb.OleDbConnection oleDbConnection1;

   private System.ComponentModel.Container components = null;

   public TableDisplay()
   {
      InitializeComponent();

      // Fill dataSet1 with data
      oleDbDataAdapter1.Fill( dataSet1, "Authors" );

      // bind data in Users table in dataSet1 to dataGrid1
      dataGrid1.SetDataBinding( dataSet1, "Authors" );
   }

   private void InitializeComponent()
   {
      this.dataSet1 = new System.Data.DataSet();
      this.oleDbDataAdapter1 =
         new System.Data.OleDb.OleDbDataAdapter();
      this.dataGrid1 = new System.Windows.Forms.DataGrid();
      this.oleDbSelectCommand1 =
         new System.Data.OleDb.OleDbCommand();
      this.oleDbInsertCommand1 =
         new System.Data.OleDb.OleDbCommand();
      this.oleDbUpdateCommand1 =
         new System.Data.OleDb.OleDbCommand();
      this.oleDbDeleteCommand1 =
         new System.Data.OleDb.OleDbCommand();
      this.oleDbConnection1 =
         new System.Data.OleDb.OleDbConnection();
      ((System.ComponentModel.ISupportInitialize)
         (this.dataSet1)).BeginInit();
      ((System.ComponentModel.ISupportInitialize)
         (this.dataGrid1)).BeginInit();
      this.SuspendLayout();
      //
      // dataSet1
      //
      this.dataSet1.DataSetName = "NewDataSet";
      this.dataSet1.Locale =
         new System.Globalization.CultureInfo("en-US");
```

Fig. 19.27 Accessing and displaying a database's data. (Part 2 of 7.)

```
         //
         // oleDbDataAdapter1
         //
         this.oleDbDataAdapter1.DeleteCommand =
            this.oleDbDeleteCommand1;
         this.oleDbDataAdapter1.InsertCommand =
            this.oleDbInsertCommand1;
         this.oleDbDataAdapter1.SelectCommand =
            this.oleDbSelectCommand1;
         this.oleDbDataAdapter1.TableMappings.AddRange(
            new System.Data.Common.DataTableMapping[] {
               new System.Data.Common.DataTableMapping(
                  "Table", "Authors",
                  new System.Data.Common.DataColumnMapping[] {
                     new System.Data.Common.DataColumnMapping(
                        "authorID", "authorID"),
                     new System.Data.Common.DataColumnMapping(
                        "firstName", "firstName"),
                     new System.Data.Common.DataColumnMapping(
                        "lastName", "lastName")})});
         this.oleDbDataAdapter1.UpdateCommand =
            this.oleDbUpdateCommand1;
         //
         // dataGrid1
         //
         this.dataGrid1.DataMember = "";
         this.dataGrid1.HeaderForeColor =
            System.Drawing.SystemColors.ControlText;
         this.dataGrid1.Location =
            new System.Drawing.Point(16, 16);
         this.dataGrid1.Name = "dataGrid1";
         this.dataGrid1.Size = new System.Drawing.Size(264, 248);
         this.dataGrid1.TabIndex = 0;
         //
         // oleDbSelectCommand1
         //
         this.oleDbSelectCommand1.CommandText =
            "SELECT authorID, firstName, lastName FROM Authors";
         this.oleDbSelectCommand1.Connection =
            this.oleDbConnection1;
         //
         // oleDbInsertCommand1
         //
         this.oleDbInsertCommand1.CommandText =
            "INSERT INTO Authors(firstName, lastName) VALUES " +
            "(?, ?)";
         this.oleDbInsertCommand1.Connection =
            this.oleDbConnection1;
         this.oleDbInsertCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter("firstName",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               "firstName"));
```

Fig. 19.27 Accessing and displaying a database's data. (Part 3 of 7.)

```
         this.oleDbInsertCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter("lastName",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               "lastName"));
         //
         // oleDbUpdateCommand1
         //
         this.oleDbUpdateCommand1.CommandText =
            "UPDATE Authors SET firstName = ?, lastName = ? WHERE" +
            " (authorID = ?) AND (firstNam" +
            "e = ? OR ? IS NULL AND firstName IS NULL) AND " +
            "(lastName = ? OR ? IS NULL AND las" +
            "tName IS NULL)";
         this.oleDbUpdateCommand1.Connection =
            this.oleDbConnection1;
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "firstName",
               System.Data.OleDb.OleDbType.VarWChar,
               50, "firstName"));
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
            "lastName",
            System.Data.OleDb.OleDbType.VarWChar, 50,
            "lastName"));
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_authorID",
               System.Data.OleDb.OleDbType.Integer, 0,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(10)), ((System.Byte)(0)),
               "authorID", System.Data.DataRowVersion.Original,
               null));
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_firstName",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
               "firstName", System.Data.DataRowVersion.Original,
               null));
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_firstName1",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
               "firstName", System.Data.DataRowVersion.Original,
               null));
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_lastName",
               System.Data.OleDb.OleDbType.VarWChar, 50,
```

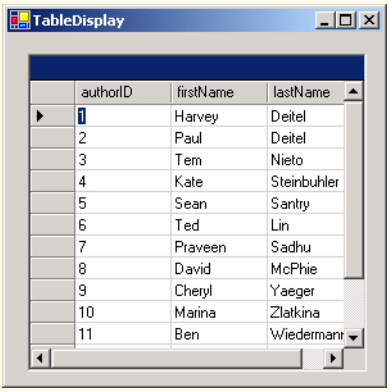
Fig. 19.27 Accessing and displaying a database's data. (Part 4 of 7.)

```
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
               "lastName", System.Data.DataRowVersion.Original,
               null));
         this.oleDbUpdateCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_lastName1",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
               "lastName", System.Data.DataRowVersion.Original,
               null));
         //
         // oleDbDeleteCommand1
         //
         this.oleDbDeleteCommand1.CommandText =
            "DELETE FROM Authors WHERE (authorID = ?) AND " +
            "(firstName = ? OR ? IS NULL AND firs" +
            "tName IS NULL) AND (lastName = ? OR ? IS NULL AND " +
            "lastName IS NULL)";
         this.oleDbDeleteCommand1.Connection =
            this.oleDbConnection1;
         this.oleDbDeleteCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_authorID",
               System.Data.OleDb.OleDbType.Integer, 0,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(10)), ((System.Byte)(0)),
               "authorID", System.Data.DataRowVersion.Original,
               null));
         this.oleDbDeleteCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_firstName",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
               "firstName", System.Data.DataRowVersion.Original,
               null));
         this.oleDbDeleteCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_firstName1",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
               "firstName", System.Data.DataRowVersion.Original,
               null));
         this.oleDbDeleteCommand1.Parameters.Add(
            new System.Data.OleDb.OleDbParameter(
               "Original_lastName",
               System.Data.OleDb.OleDbType.VarWChar, 50,
               System.Data.ParameterDirection.Input, false,
               ((System.Byte)(0)), ((System.Byte)(0)),
```

Fig. 19.27 Accessing and displaying a database's data. (Part 5 of 7.)

```
            "lastName", System.Data.DataRowVersion.Original,
            null));
      this.oleDbDeleteCommand1.Parameters.Add(
         new System.Data.OleDb.OleDbParameter(
            "Original_lastName1",
            System.Data.OleDb.OleDbType.VarWChar, 50,
            System.Data.ParameterDirection.Input, false,
            ((System.Byte)(0)), ((System.Byte)(0)),
            "lastName", System.Data.DataRowVersion.Original,
            null));
      //
      // oleDbConnection1
      //
      this.oleDbConnection1.ConnectionString =
         @"Provider=Microsoft.Jet.OLEDB.4.0;Password="""";" +
         @"User ID=Admin;Data Source=C:\Books\2001\csphtp1\" +
         @"csphtp1_examples\ch19\Books.mdb;Mode=Share " +
         @"Deny None;Extended Properties="""";Jet OLEDB:" +
         @"System database="""";Jet OLEDB:Registry " +
         @"Path="""";Jet OLEDB:Database Password="""";" +
         @"Jet OLEDB:Engine Type=5;Jet OLEDB:Database " +
         @"Locking Mode=1;Jet OLEDB:Global Partial Bulk " +
         @"Ops=2;Jet OLEDB:Global Bulk Transactions=1;Jet " +
         @"OLEDB:New Database Password="""";Jet OLEDB:" +
         @"Create System Database=False;Jet OLEDB:Encrypt " +
         @"Database=False;Jet OLEDB:Don't Copy Locale on " +
         @"Compact=False;Jet OLEDB:Compact Without Replica " +
         @"Repair=False;Jet OLEDB:SFP=False";
      //
      // TableDisplay
      //
      this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      this.ClientSize = new System.Drawing.Size(292, 273);
      this.Controls.AddRange(
         new System.Windows.Forms.Control[] {
            this.dataGrid1});
      this.Name = "TableDisplay";
      this.Text = "TableDisplay";
      ((System.ComponentModel.ISupportInitialize)
         (this.dataSet1)).EndInit();
      ((System.ComponentModel.ISupportInitialize)
         (this.dataGrid1)).EndInit();
      this.ResumeLayout(false);

   }   // end of InitializeComponent

   [STAThread]
   static void Main()
   {
      Application.Run( new TableDisplay() );
   }
}
```

Fig. 19.27 Accessing and displaying a database's data. (Part 6 of 7.)



	authorID	firstName	lastName
▶	1	Harvey	Deitel
	2	Paul	Deitel
	3	Tem	Nieto
	4	Kate	Steinbuhler
	5	Sean	Santry
	6	Ted	Lin
	7	Praveen	Sadhu
	8	David	McPhie
	9	Cheryl	Yaeger
	10	Marina	Zlatkina
	11	Ben	Wiedermanr

Fig. 19.27 Accessing and displaying a database's data. (Part 7 of 7.)

Next, drag an ***`OleDbDataAdapter`*** from the **Toolbox**'s ***Data*** group onto the Windows Form designer. This displays the **Data Adapter Configuration Wizard** for configuring the **`OleDbDataAdapter`** instance with a custom query for populating a **`DataSet`**. Click **Next** to select a connection to use. Select the connection created in the previous step from the drop-down list and click **Next**. The resulting screen allows us to choose how the **`OleDbDataAdapter`** should access the database. Keep the default **Use SQL Statement** option and then click **Next**. Click the **Query Builder** button, select the **`Authors`** table from the **Add** menu and **`Close`** that menu. Place a check mark in the ***All Columns** box from the **Authors** window. Notice how that particular window lists all columns of the **`Authors`** table.

Next, we must create a **`DataSet`** to store the query results. To do so, drag **`DataSet`** from the **Data** group in the **Toolbox**. This displays the **Add DataSet** window. Choose the **Untyped DataSet (no schema)**, because the query with which we populate the **`DataSet`** dictates the **`DataSet`**'s *schema*, or structure.

Figure 19.27 shows all of the code generated by Visual Studio. Normally, we omit this code, because it usually only contains GUI related code. In this case, however, the code contains database functionality that we must discuss. Furthermore, we have left the default naming conventions of Visual Studio in this example to demonstrate the exact format of the auto-generated code that Visual Studio creates. Normally, we would change these names to conform to our programming conventions and style. The code generated by Visual Studio has also been formatted for presentation purposes.

Good Programming Practice 19.2

Use clear, descriptive variable names in code. This makes programs easier to understand.

Lines 233–247 initialize the **`oleDbConnection`** for this program. The **`ConnectionString`** property specifies the path to the database file on the computer's hard drive.

An instance of class **`OleDbDataAdapter`** populates the **`DataSet`** in this example with data from the **`Books`** database. The instance properties ***`DeleteCommand`*** (lines 66–67), ***`InsertCommand`*** (lines 68–69), ***`SelectCommand`*** (lines 70–71) and ***`Update-`***

`Command` (lines 83–84) are **`OleDbCommand`** objects that specify how the **`OleDbDataAdapter`** deletes, inserts, selects and updates data in the database, respectively.

Each **`OleDbCommand`** object must have an **`OleDbConnection`** through which the **`OleDbCommand`** can communicate with the database. Property **`Connection`** is set to the **`OleDbConnection`** to the **`Books`** database. For **`oleDbUpdateCommand1`**, lines 128–129 set the **`Connection`** property, and lines 122–127 set the **`CommandText`**.

Although Visual Studio generates most of this program's code, we enter code in the **`TableDisplay`** constructor (lines 25–34) for populating **`dataSet1`** using an **`OleDbDataAdapter`**. Line 30 calls **`OleDbDataAdapter`** method ***`Fill`*** to retrieve information from the database associated with the **`OleDbConnection`**, placing the information in the **`DataSet`** provided as an argument. The second argument to this method is a **`string`** that specifies the name of the table in the database from which to **`Fill`** the **`DataSet`**.

Line 33 invokes **`DataGrid`** method **`SetDataBinding`** to bind the **`DataGrid`** to a data source. The first argument is the **`DataSet`**—in this case, **`dataSet1`**—whose data the **`DataGrid`** should display. The second argument is a **`string`** representing the name of the table within the data source we want to bind to the **`DataGrid`**. Once this line executes, the **`DataGrid`** is filled with the information in the **`DataSet`**—the number of rows and number of columns are set from the information in **`dataSet1`**.

19.6.2 Querying the `Books` Database

The example in Fig. 19.28 demonstrates how to execute SQL **`SELECT`** statements on database **`Books.mdb`** and display the results. Although Fig. 19.28 uses only **`SELECT`** statements to query the data, the same program could be used to execute many different SQL statements if we made a few minor modifications.

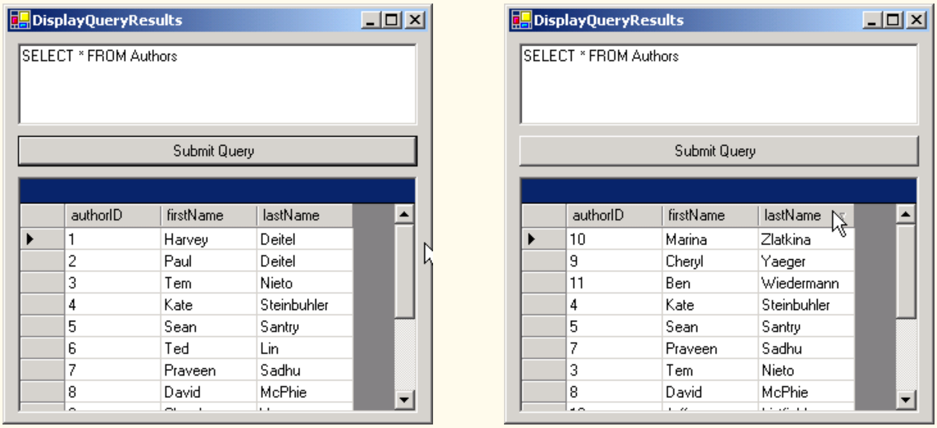
Method **`submitButton_Click`** is the key part of this program. When the program invokes this event handler, lines 47–48 assign the **`SELECT`** query **`string`** to **`OleDbDataAdapter`**'s **`SelectCommand`** property. This **`string`** is parsed into an SQL query and executed on the database via the **`OleDbDataAdapter`**'s **`Fill`** method (line 55). As we discussed in the previous section, method **`Fill`** places data from the database into **`dataSet1`**.

```
 1  // Fig. 19.28: DisplayQueryResults.cs
 2  // Displays the contents of the authors database.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10
11  public class DisplayQueryResults : System.Windows.Forms.Form
12  {
13     private System.Data.OleDb.OleDbConnection oleDbConnection1;
14     private System.Data.DataSet dataSet1;
```

Fig. 19.28 Execute SQL statements on a database. (Part 1 of 3.)

```
   private System.Data.OleDb.OleDbDataAdapter oleDbDataAdapter1;
   private System.Data.OleDb.OleDbCommand oleDbSelectCommand1;
   private System.Data.OleDb.OleDbCommand oleDbInsertCommand1;
   private System.Data.OleDb.OleDbCommand oleDbUpdateCommand1;
   private System.Data.OleDb.OleDbCommand oleDbDeleteCommand1;
   private System.Windows.Forms.TextBox queryTextBox;
   private System.Windows.Forms.Button submitButton;
   private System.Windows.Forms.DataGrid dataGrid1;
   private System.ComponentModel.Container components = null;

   public DisplayQueryResults()
   {

      InitializeComponent();
   }

   // Visual Studio.NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new DisplayQueryResults() );
   }

   // perform SQL query on data
   private void submitButton_Click( object sender,
      System.EventArgs e )
   {
      try
      {
         // set SQL query to what user
         // input into queryTextBox
         oleDbDataAdapter1.SelectCommand.CommandText =
            queryTextBox.Text;

         // clear DataSet from previous operation
         dataSet1.Clear();

         // Fill data set with information that results
         // from SQL query
         oleDbDataAdapter1.Fill( dataSet1, "Authors" );

         // bind DataGrid to contents of DataSet
         dataGrid1.SetDataBinding( dataSet1, "Authors" );
      }

      catch ( System.Data.OleDb.OleDbException oleException )
      {
         MessageBox.Show( "Invalid query" );
      }

   }  // end of submitButton_Click
}
```

Fig. 19.28 Execute SQL statements on a database. (Part 2 of 3.)



SELECT * FROM Authors

	authorID	firstName	lastName
▶	1	Harvey	Deitel
	2	Paul	Deitel
	3	Tem	Nieto
	4	Kate	Steinbuhler
	5	Sean	Santry
	6	Ted	Lin
	7	Praveen	Sadhu
	8	David	McPhie

SELECT * FROM Authors

	authorID	firstName	lastName
▶	10	Marina	Zlatkina
	9	Cheryl	Yaeger
	11	Ben	Wiedermann
	4	Kate	Steinbuhler
	5	Sean	Santry
	7	Praveen	Sadhu
	3	Tem	Nieto
	8	David	McPhie

Fig. 19.28 Execute SQL statements on a database. (Part 3 of 3.)

Common Programming Error 19.9

If a **`DataSet`** *has been* **`Fill`***ed at least once, forgetting to call a* **`DataSet`***'s* **`Clear`** *method before calling the* **`Fill`** *method again will lead to logic errors.*

To display, or redisplay, contents in the **`DataGrid`**, use method **`SetDataBinding`**. The first argument is the datasource to be displayed in the table—a **`DataSet`**, in this case. The second argument is the **`string`** name of the datasource member to be displayed (line 58). Readers can try entering their own queries in the text box and then pressing the **Submit Query** button to execute the query.

19.7 Programming with ADO.NET: Modifying a DBMS

Our next example implements a simple address-book application that enables the user to insert, locate and update records in the Microsoft Access database **`Addressbook`**.

The **`Addressbook`** application (Fig. 19.29) provides a GUI enabling users to execute SQL statements on the database. Earlier in the chapter, we presented examples demonstrating the use of **`SELECT`** statements to query a database. Here, that same functionality is provided.

```
// Fig. 19.29: AddressBook.cs
// Using SQL statements to manipulate a database.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

```

Fig. 19.29 Modifying a database. (Part 1 of 8.)

```
public class AddressBook : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox faxTextBox;
   private System.Windows.Forms.TextBox homeTextBox;
   private System.Windows.Forms.TextBox firstTextBox;
   private System.Windows.Forms.TextBox stateTextBox;
   private System.Windows.Forms.TextBox idTextBox;
   private System.Windows.Forms.TextBox lastTextBox;
   private System.Windows.Forms.TextBox postalTextBox;
   private System.Windows.Forms.TextBox addressTextBox;
   private System.Windows.Forms.TextBox cityTextBox;
   private System.Windows.Forms.TextBox countryTextBox;
   private System.Windows.Forms.TextBox emailTextBox;
   private System.Data.DataSet dataSet1;
   private System.Data.OleDb.OleDbDataAdapter oleDbDataAdapter1;
   private System.Data.OleDb.OleDbCommand oleDbSelectCommand1;
   private System.Data.OleDb.OleDbCommand oleDbInsertCommand1;
   private System.Data.OleDb.OleDbCommand oleDbUpdateCommand1;
   private System.Data.OleDb.OleDbCommand oleDbDeleteCommand1;
   private System.Data.OleDb.OleDbConnection oleDbConnection1;
   private System.Windows.Forms.TextBox statusTextBox;
   private System.Windows.Forms.Label addressLabel;
   private System.Windows.Forms.Label cityLabel;
   private System.Windows.Forms.Label stateLabel;
   private System.Windows.Forms.Label idLabel;
   private System.Windows.Forms.Label firstLabel;
   private System.Windows.Forms.Label lastLabel;
   private System.Windows.Forms.Label postalLabel;
   private System.Windows.Forms.Label countryLabel;
   private System.Windows.Forms.Label emailLabel;
   private System.Windows.Forms.Button clearButton;
   private System.Windows.Forms.Button helpButton;
   private System.Windows.Forms.Button findButton;
   private System.Windows.Forms.Button addButton;
   private System.Windows.Forms.Button updateButton;
   private System.Windows.Forms.Label faxLabel;
   private System.Windows.Forms.Label homeLabel;
   private System.ComponentModel.Container components = null;

   public AddressBook()
   {
      InitializeComponent();
      oleDbConnection1.Open();
   }

   // Visual Studio.NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new AddressBook() );
   }

```

Fig. 19.29 Modifying a database. (Part 2 of 8.)

```
   private void findButton_Click( object sender,
         System.EventArgs e )
   {
      try
      {
         if ( lastTextBox.Text != "" )
         {
            // clear DataSet from last operation
            dataSet1.Clear();

            // create SQL query to find contact with
            // specified last name
            oleDbDataAdapter1.SelectCommand.CommandText =
               "SELECT * FROM addresses WHERE lastname = '" +
               lastTextBox.Text + "'";

            // fill dataSet1 with rows resulting from
            // query
            oleDbDataAdapter1.Fill( dataSet1 );

            // display information
            Display( dataSet1 );
            statusTextBox.Text += "\r\nQuery successful\r\n";
         }
         else
            lastTextBox.Text =
               "Enter last name here then press Find";
      }

      catch ( System.Data.OleDb.OleDbException oleException )
      {
         Console.WriteLine( oleException.StackTrace );
         statusTextBox.Text += oleException.ToString();
      }

      catch ( InvalidOperationException invalidException )
      {
         MessageBox.Show( invalidException.Message );
      }

   }  // end of findButton_Click

   private void addButton_Click( object sender, System.EventArgs e )
   {
      try
      {
         if ( lastTextBox.Text != "" && firstTextBox.Text != "" )
         {
            // create SQL query to insert row
            oleDbDataAdapter1.InsertCommand.CommandText =
               "INSERT INTO addresses (" +
               "firstname, lastname, address, city, " +
               "stateorprovince, postalcode, country, " +
```

Fig. 19.29 Modifying a database. (Part 3 of 8.)

```
                  "emailaddress, homephone, faxnumber" +
                  ") VALUES ('" +
                  firstTextBox.Text + "', '" +
                  lastTextBox.Text + "', '" +
                  addressTextBox.Text + "', '" +
                  cityTextBox.Text + "', '" +
                  stateTextBox.Text + "', '" +
                  postalTextBox.Text + "', '" +
                  countryTextBox.Text + "', '" +
                  emailTextBox.Text + "', '" +
                  homeTextBox.Text + "', '" +
                  faxTextBox.Text + "')";

               // notify user that query is being sent
               statusTextBox.Text += "\r\nSending query: " +
                  oleDbDataAdapter1.InsertCommand.CommandText +
                  "\r\n" ;

               // send query
               oleDbDataAdapter1.InsertCommand.ExecuteNonQuery();

               statusTextBox.Text += "\r\nQuery successful\r\n";
            }
            else
               statusTextBox.Text += "\r\nEnter at least first " +
                  "and last name then press Add\r\n";
         }

         catch ( System.Data.OleDb.OleDbException oleException )
         {
            Console.WriteLine( oleException.StackTrace );
            statusTextBox.Text += oleException.ToString();
         }

      }  // end of addButton_Click

      private void updateButton_Click( object sender,
         System.EventArgs e )
      {
         try
         {
            // make sure users have found record
            // they wish to update
            if ( idTextBox.Text != "" )
            {
               // set SQL query to update all fields in
               // table where id number matches id
               // in idTextBox
               oleDbDataAdapter1.UpdateCommand.CommandText =
                  "UPDATE addresses SET " +
                  "firstname ='" + firstTextBox.Text +
                  "', lastname='" + lastTextBox.Text +
                  "', address='" + addressTextBox.Text +
```

Fig. 19.29 Modifying a database. (Part 4 of 8.)

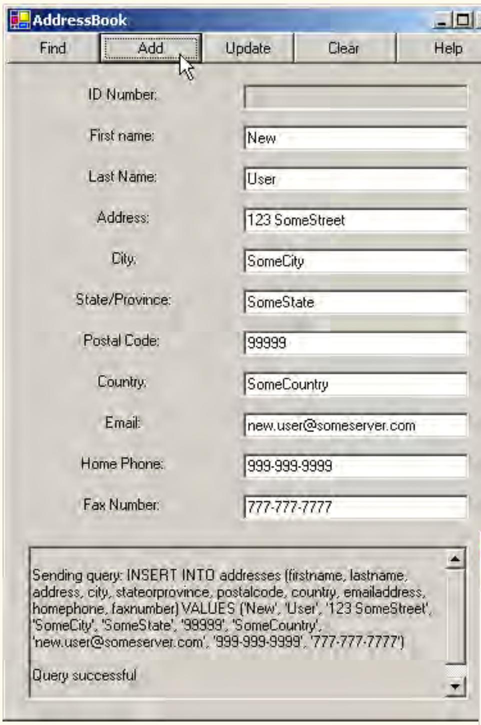
```
               "', city='" + cityTextBox.Text +
               "', stateorprovince='" + stateTextBox.Text +
               "', postalcode='" + postalTextBox.Text +
               "', country='" + countryTextBox.Text +
               "', emailaddress='" + emailTextBox.Text +
               "', homephone='" + homeTextBox.Text +
               "', faxnumber='" + faxTextBox.Text +
               "' WHERE id=" + idTextBox.Text;

            // notify user that query is being set
            statusTextBox.Text += "\r\nSending query: " +
               oleDbDataAdapter1.UpdateCommand.CommandText +
               "\r\n";

            // execute query
            oleDbDataAdapter1.UpdateCommand.ExecuteNonQuery();

            statusTextBox.Text += "\r\nQuery successful\r\n";
         }
         else
            statusTextBox.Text += "\r\nYou may only update " +
               "an existing record. Use Find to locate the" +
               "record, then modify the information and " +
               "press Update.\r\n";
      }

      catch ( System.Data.OleDb.OleDbException oleException )
      {
         Console.WriteLine( oleException.StackTrace );
         statusTextBox.Text += oleException.ToString();
      }

   } // end of updateButton_Click

   private void clearButton_Click( object sender,
         System.EventArgs e )
   {
      idTextBox.Clear();
      ClearTextBoxes();
   }

   private void helpButton_Click( object sender,
         System.EventArgs e )
   {
      statusTextBox.AppendText(
         "\r\nClick Find to locate a record\r\n" +
         "Click Add to insert a new record.\r\n" +
         "Click Update to update the information in a record "
         + "\r\nClick Clear to empty the textboxes" );
   }

   private void Display( DataSet dataSet )
   {
```

Fig. 19.29 Modifying a database. (Part 5 of 8.)

```
      try
      {
         // get first DataTable--there always will be one
         DataTable dataTable = dataSet.Tables[ 0 ];

         if ( dataTable.Rows.Count != 0 )
         {
            int recordNumber = ( int ) dataTable.Rows[ 0 ][ 0 ];

            idTextBox.Text = recordNumber.ToString();
            firstTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 1 ];
            lastTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 2 ];
            addressTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 3 ];
            cityTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 4 ];
            stateTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 5 ];
            postalTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 6 ];
            countryTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 7 ];
            emailTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 8 ];
            homeTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 9 ];
            faxTextBox.Text =
               ( string ) dataTable.Rows[ 0 ][ 10 ];
         }

         else
            statusTextBox.Text += "\r\nNo record found\r\n";
      }

      catch( System.Data.OleDb.OleDbException oleException )
      {
         Console.WriteLine( oleException.StackTrace );
         statusTextBox.Text += oleException.ToString();
      }

   } // end Display

   private void ClearTextBoxes()
   {
      firstTextBox.Clear();
      lastTextBox.Clear();
      addressTextBox.Clear();
      cityTextBox.Clear();
      stateTextBox.Clear();
      postalTextBox.Clear();
      countryTextBox.Clear();
```

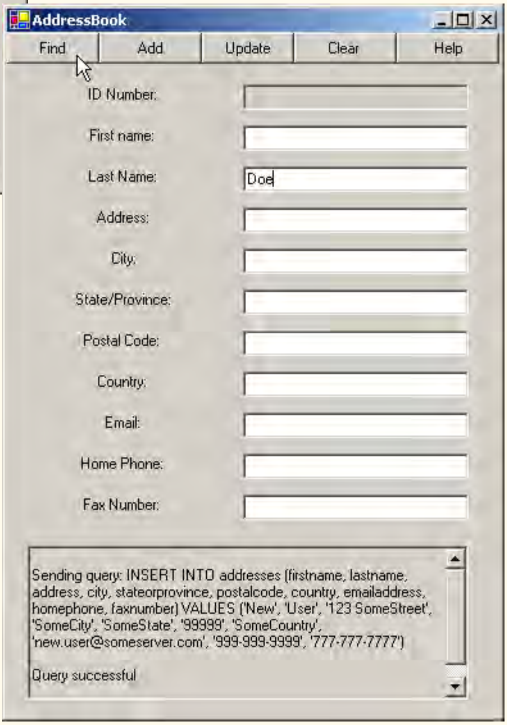
Fig. 19.29 Modifying a database. (Part 6 of 8.)

```
         emailTextBox.Clear();
         homeTextBox.Clear();
         faxTextBox.Clear();
      }
   }
```



ID Number:
First name: New
Last Name: User
Address: 123 SomeStreet
City: SomeCity
State/Province: SomeState
Postal Code: 99999
Country: SomeCountry
Email: new.user@someserver.com
Home Phone: 999-999-9999
Fax Number: 777-777-7777

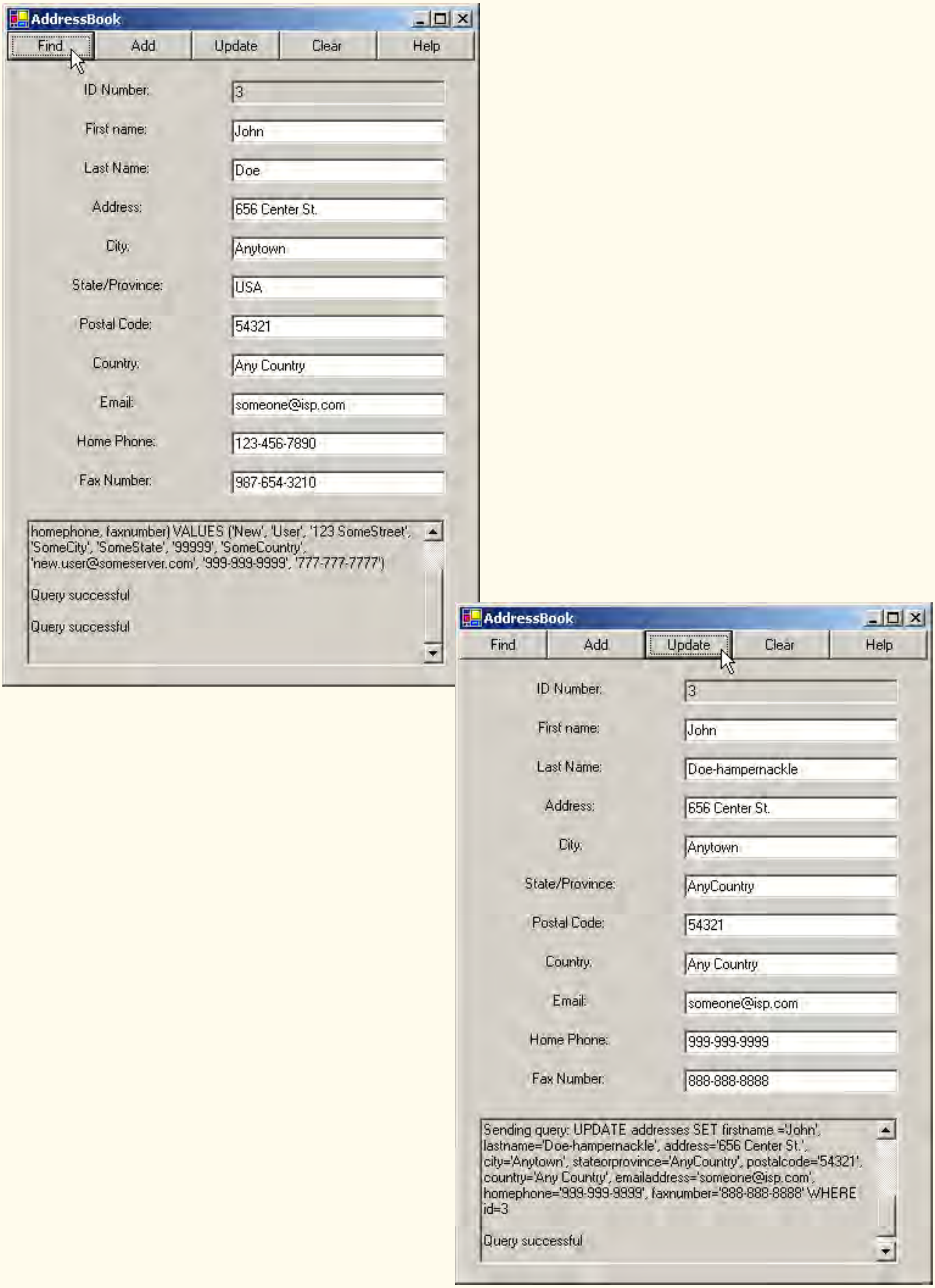
```
Sending query: INSERT INTO addresses (firstname, lastname,
address, city, stateorprovince, postalcode, country, emailaddress,
homephone, faxnumber) VALUES ('New', 'User', '123 SomeStreet',
'SomeCity', 'SomeState', '99999', 'SomeCountry',
'new.user@someserver.com', '999-999-9999', '777-777-7777')

Query successful
```



ID Number:
First name:
Last Name: Doe
Address:
City:
State/Province:
Postal Code:
Country:
Email:
Home Phone:
Fax Number:

```
Sending query: INSERT INTO addresses (firstname, lastname,
address, city, stateorprovince, postalcode, country, emailaddress,
homephone, faxnumber) VALUES ('New', 'User', '123 SomeStreet',
'SomeCity', 'SomeState', '99999', 'SomeCountry',
'new.user@someserver.com', '999-999-9999', '777-777-7777')

Query successful
```

Fig. 19.29 Modifying a database. (Part 7 of 8.)



ID Number: 3
First name: John
Last Name: Doe
Address: 656 Center St.
City: Anytown
State/Province: USA
Postal Code: 54321
Country: Any Country
Email: someone@isp.com
Home Phone: 123-456-7890
Fax Number: 987-654-3210

```
homephone, faxnumber) VALUES ('New', 'User', '123 SomeStreet',
'SomeCity', 'SomeState', '99999', 'SomeCountry',
'new.user@someserver.com', '999-999-9999', '777-777-7777')

Query successful

Query successful
```

ID Number: 3
First name: John
Last Name: Doe-hampernackle
Address: 656 Center St.
City: Anytown
State/Province: AnyCountry
Postal Code: 54321
Country: Any Country
Email: someone@isp.com
Home Phone: 999-999-9999
Fax Number: 888-888-8888

```
Sending query: UPDATE addresses SET firstname ='John',
lastname='Doe-hampernackle', address='656 Center St.',
city='Anytown', stateorprovince='AnyCountry', postalcode='54321',
country='Any Country', emailaddress='someone@isp.com',
homephone='999-999-9999', faxnumber='888-888-8888' WHERE
id=3

Query successful
```

Fig. 19.29 Modifying a database. (Part 8 of 8.)

Event handler **`findButton_Click`** performs the **`SELECT`** query on the database for the record associated with the **`string`** in **`lastTextBox`**. This represents the last name of the person whose record the user wishes to retrieve. Line 72 invokes method **`Clear`** of class **`DataSet`** to empty the **`DataSet`** of any prior data. Lines 76–78 then modify the text of the SQL query to perform the appropriate **`SELECT`** operation. This statement is executed by the **`OleDbDataAdapter`** method **`Fill`** (line 82), which is passed the **`DataSet`** as an argument. Finally, the **`TextBox`**es are updated with a call to method **`Display`** (line 85).

Methods **`addButton_Click`** and **`updateButton_Click`** perform **`INSERT`** and **`UPDATE`** operations, respectively. Each method uses members of class **`OleDbCommand`** to perform operations on a database. The instance properties **`InsertCommand`** and **`UpdateCommand`** of class **`OleDbDataAdapter`** are instances of class **`OleDbCommand`**.

Property **`CommandText`** of class **`OleDbCommand`** is a **`string`** representing the SQL statement that the **`OleDbCommand`** object executes. Method **`addButton_Click`** sets this property of **`InsertCommand`** to execute the appropriate **`INSERT`** statement on the database (lines 113–128). Method **`updateButton_Click`** sets this property of **`UpdateCommand`** to execute the appropriate **`UPDATE`** statement on the database (lines 165–177).

Method ***`ExecuteNonQuery`*** of class **`OleDbCommand`** performs the action specified by **`CommandText`**. Hence, the **`INSERT`** statement defined by **`oleDbDataAdapter1.InsertCommand.CommandText`** in event handler **`addButton_Click`** is executed when line 136 invokes method **`oleDbDataAdapter1.InsertCommand.ExecuteNonQuery`**. Similarly, the **`UPDATE`** statement defined by **`oleDbDataAdapter1.DeleteCommand.CommandText`** in **`updateButton_Click`** event handler is executed by **`oleDbDataAdapter1.UpdateCommand.ExecuteNonQuery`** (line 185).

Method **`Display`** (lines 221–265) updates the user interface with data from the newly retrieved address-book record. Line 226 obtains a **`DataTable`** from the **`DataSet`**'s **`Tables`** collection. This **`DataTable`** contains the results of our SQL query. Line 228 determines whether the query returned any rows. The **`Rows`** property in class **`DataTable`** provides access to all records retrieved by the query. The **`Rows`** property is similar to a two-dimensional rectangular array. Line 230 retrieves the field with index *0, 0* (i.e., the first record's first column of data) and stores the value in variable **`recordNumber`**. Lines 232–252 then retrieve the remaining fields of data from the **`DataTable`** to populate the user interface.

When clicked, the application's **Help** button prints instructions in the console at the bottom of the application window (lines 214–218). The event handler for this button is **`helpButton_Click`**. The **`Clear`** button clears the text from the **`TextBox`**es. This event handler is defined in the method **`clearButton_Click`** and uses the utility method **`ClearTextBoxes`** (line 208).

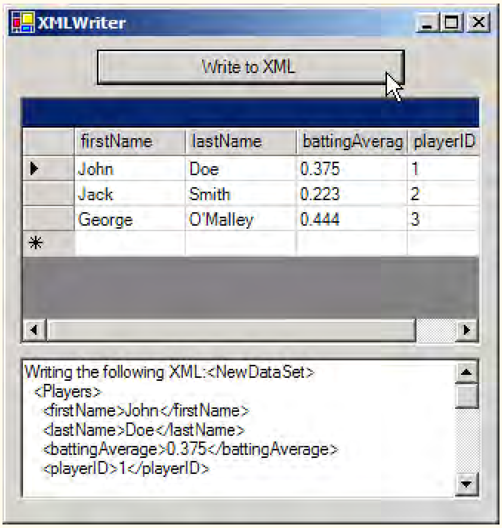
19.8 Reading and Writing XML Files

A powerful feature of ADO .NET is its ability to convert data stored in a datasource to XML. Class **`DataSet`** of namespace **`System.Data`** provides methods ***`WriteXml`***, ***`ReadXml`*** and ***`GetXml`***, which enable developers to create XML documents from datasources and to convert data from XML into datasources. The application in Fig. 19.30 populates a **`DataSet`** with statistics about baseball players and then writes the data to a file as XML. The application also displays the XML in a **`TextBox`**.

```
// Fig. 19.30 XMLWriter.cs
// Demonstrates generating XML from an ADO .NET DataSet.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class DatabaseXMLWriter : System.Windows.Forms.Form
{
   private System.Data.OleDb.OleDbConnection baseballConnection;
   private System.Data.OleDb.OleDbDataAdapter playersDataAdapter;
   private System.Data.OleDb.OleDbCommand oleDbSelectCommand1;
   private System.Data.OleDb.OleDbCommand oleDbInsertCommand1;
   private System.Data.OleDb.OleDbCommand oleDbUpdateCommand1;
   private System.Data.OleDb.OleDbCommand oleDbDeleteCommand1;
   private System.Data.DataSet playersDataSet;
   private System.Windows.Forms.DataGrid playersDataGrid;
   private System.Windows.Forms.Button writeButton;
   private System.Windows.Forms.TextBox outputTextBox;
   private System.ComponentModel.Container components = null;

   public DatabaseXMLWriter()
   {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();

      // open database connection
      baseballConnection.Open();

      // fill DataSet with data from OleDbDataAdapter
      playersDataAdapter.Fill( playersDataSet, "Players" );

      // bind DataGrid to DataSet
      playersDataGrid.SetDataBinding( playersDataSet, "Players" );

   }

   // Visual Studio .NET generated code

   // main entry point for application.
   [STAThread]
   static void Main()
   {
      Application.Run( new DatabaseXMLWriter() );
   }
```

Fig. 19.30 Application that writes an XML representation of a **`DataSet`** to a file.

```
51
52     // write XML representation of DataSet when button is clicked
53     private void writeButton_Click(
54        object sender, System.EventArgs e)
55     {
56        // write XML representation of DataSet to file
57        playersDataSet.WriteXml( "Players.xml" );
58
59        // display XML in TextBox
60        outputTextBox.Text += "Writing the following XML:\n\n" +
61           playersDataSet.GetXml() + "\n\n";
62
63     }
64  }
```



	firstName	lastName	battingAverag	playerID
▶	John	Doe	0.375	1
	Jack	Smith	0.223	2
	George	O'Malley	0.444	3
*				

```
Writing the following XML:<NewDataSet>
 <Players>
  <firstName>John</firstName>
  <lastName>Doe</lastName>
  <battingAverage>0.375</battingAverage>
  <playerID>1</playerID>
```

Fig. 19.30 Application that writes an XML representation of a `DataSet` to a file.

The `DatabaseXMLWriter` constructor (lines 25–41) establishes a connection to the `Baseball` database in line 33. Line 36 then calls method `Fill` of class `OleDbDataAdapter` to populate `playersDataSet` with data from the `Players` table in the `Baseball` database. Line 39 binds `playersDataGrid` to `playersDataSet` to display the information to the user.

Method `writeButton_Click` defines the event handler for the **Write to XML** button. When the user clicks this button, line 57 invokes `DataSet` method `WriteXml`, which generates an XML representation of the data contained in the `DataSet` and writes the XML to the specified file. Figure 19.31 depicts this XML representation. Each `Players` element represents a record in the `Players` table. The `firstName`, `lastName`, `battingAverage` and `playerID` elements correspond to the fields of the same names in the `Players` table. Method `GetXml` returns a `string` representing the `DataSet`'s data in XML form. Lines 60–61 append the XML `string` to `outputTextBox`.

```xml
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
   <Players>
      <firstName>John</firstName>
      <lastName>Doe</lastName>
      <battingAverage>0.375</battingAverage>
      <playerID>1</playerID>
   </Players>

   <Players>
      <firstName>Jack</firstName>
      <lastName>Smith</lastName>
      <battingAverage>0.223</battingAverage>
      <playerID>2</playerID>
   </Players>

   <Players>
      <firstName>George</firstName>
      <lastName>O'Malley</lastName>
      <battingAverage>0.444</battingAverage>
      <playerID>3</playerID>
   </Players>
</NewDataSet>
```

Fig. 19.31 XML document generated from `DataSet` in `DatabaseXMLWriter`.

In this chapter, we discussed the fundamentals of Structured Query Language (SQL) and C#'s database capabilities. We learned that C# programmers communicate with databases and manipulate their data through *Microsoft ActiveX Data Objects™* (ADO), *ADO .NET*. In the next chapter we discuss ASP .NET Web Forms. Web Forms allow programmers to develop dynamic Web content using databases and ASP .NET features.

SUMMARY

- A database is an integrated collection of data. A database management system (DBMS) provides mechanisms for storing and organizing data.
- Today's most popular database systems are relational databases.
- A language called Structured Query Language (SQL) is used almost universally with relational-database systems to perform queries and manipulate data.
- A programming language connects to, and interacts with, relational databases via an interface—software that facilitates communications between a database management system and a program.
- C# programmers communicate with databases and manipulate their data via ADO .NET.
- A relational database is composed of tables. A row of a table is called a record.
- A primary key is a field that contains unique data, or data that is not duplicated in other records of that table.
- Each column in a table represents a different field (or attribute).
- A primary key can be composed of more than one column (or field) in the database.
- SQL provides a complete set of commands, enabling programmers to define complex queries to select data from a table. The results of a query commonly are called result sets (or record sets).

- A one-to-many relationship between tables indicates that a record in one table can have many corresponding records in a separate table.
- A foreign key is a field for which every entry in one table has a unique value in another table and where the field in the other table is the primary key for that table.
- The basic format for a `SELECT` query is:

  ```
  SELECT * FROM tableName
  ```

  where the asterisk (`*`) indicates that all columns from *tableName* should be selected, and *tableName* specifies the table in the database from which the data will be selected.
- To select specific fields from a table, replace the asterisk (`*`) with a comma-separated list of the field names to select.
- Programmers process result sets by knowing in advance the order of the fields in the result set. Specifying the field names to select guarantees that the fields are returned in the specified order, even if the actual order of the fields in the database table(s) changes.
- The optional `WHERE` clause in a `SELECT` query specifies the selection criteria for the query. The simplest format for a `SELECT` query with selection criteria is:

  ```
  SELECT fieldName1, fieldName2, … FROM tableName WHERE criteria
  ```

- The `WHERE` clause condition can contain operators `<`, `>`, `<=`, `>=`, `=`, `<>` and `LIKE`. Operator `LIKE` is used for pattern matching with wildcard characters asterisk (`*`) and question mark (`?`).
- A pattern `string` containing an asterisk character (`*`) searches for strings in which zero or more characters appear in the asterisk character's location in the pattern.
- A pattern string containing a question mark (`?`) searches for strings in which exactly one character appears in the question mark's position in the pattern.
- The results of a query can be arranged in ascending or descending order via the optional `ORDER BY` clause. The simplest form of an `ORDER BY` clause is:

  ```
  SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field ASC
  SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field DESC
  ```

  where `ASC` specifies ascending order, `DESC` specifies descending order and *field* specifies the field to be sorted. The default sorting order is ascending, so `ASC` is optional.
- An `ORDER BY` clause also can sort records by multiple fields. Such queries are written in the form:

  ```
  ORDER BY field1 sortingOrder, field2 sortingOrder, …
  ```

- The `WHERE` and `ORDER BY` clauses can be combined in one query.
- A join merges records from two or more tables by testing for matching values in a field that is common to both tables. The simplest format of a join is:

  ```
  SELECT fieldName1, fieldName2, …
     FROM table1, table2
     WHERE table1.fieldName = table2.fieldName
  ```

  in which the `WHERE` clause specifies the fields from each table that should be compared to determine which records are joined. These fields normally represent the primary key in one table and the corresponding foreign key in another table.
- If an SQL statement uses fields that have the same name in multiple tables, the statement must fully qualify the field name by preceding it with its table name and the dot operator (`.`).
- An `INSERT` statement inserts a new record in a table. The simplest form for this statement is:

  ```
  INSERT INTO tableName ( fieldName1, fieldName2, …, fieldNameN )
     VALUES ( value1, value2, …, valueN )
  ```

where *tableName* is the table in which to insert the record. The *tableName* is followed by a comma-separated list of field names in parentheses. The list of field names is followed by the SQL keyword **`VALUES`** and a comma-separated list of values in parentheses.

- SQL statements use a single quote (**`'`**) as a delimiter for strings. To specify a string containing a single quote in an SQL statement, the single quote must be escaped with another single quote.
- An **`UPDATE`** statement modifies data in a table. The simplest form for an **`UPDATE`** statement is:

```
UPDATE tableName
   SET fieldName1 = value1, fieldName2 = value2, …, fieldNameN = valueN
   WHERE criteria
```

  where *tableName* is the table in which to update a record (or records). The *tableName* is followed by keyword **`SET`** and a comma-separated list of field-name/value pairs, written in the format *fieldName* **`=`** *value*. The **`WHERE`** *criteria* determine the record(s) to update.
- A **`DELETE`** statement removes data from a table. The simplest form for a **`DELETE`** statement is:

```
DELETE FROM tableName WHERE criteria
```

  where *tableName* is the table from which to delete a record (or records). The **`WHERE`** *criteria* determine which record(s) to delete.
- **`System.Data`**, **`System.Data.OleDb`** and **`System.Data.SqlClient`** are the three main namespaces in ADO .NET.
- Class **`DataSet`** is from the **`System.Data`** namespace. Instances of this class represent in-memory caches of data.
- The advantage of using class **`DataSet`** is that it is a way to modify the contents of a datasource without having to maintain an active connection.
- One approach to ADO .NET programming uses **`OleDbCommand`** of the **`System.Data.OleDb`** namespace. In this approach, SQL statements are executed directly on the datasource.
- Use the **Add Connection** option to create a database connection in the **Data Link Properties** window.
- Use the **Data Adapter Configuration Wizard** to set up an **`OleDbDataAdapter`** and generate queries.
- If a **`DataSet`** needs to be named, use the instance property **`DataSetName`**.
- **`OleDbCommands`** commands are what the **`OleDbDataAdapter`** executes on the database in the form of SQL queries.
- **`DataColumnMapping`**s converts data from a database to a **`DataSet`**, and vice versa.
- Instance property **`Parameters`** of class **`OleDbCommand`** is a collection of **`OleDbParameter`** objects. Adding them to an **`OleDbCommand`** is an optional way to add parameters in a command, instead of creating a lengthy, complex command string.
- **`OleDbCommand`** instance property **`Connection`** is set to the **`OleDbConnection`** that the command will be executed on, and the instance property **`CommandText`** is set to the SQL query that will be executed on the database.
- **`OleDbDataAdapter`** method **`Fill`** retrieves information from the database associated with the **`OleDbConnection`** and places this information in the **`DataSet`** provided as an argument.
- **`DataGrid`** method **`SetDataBinding`** binds a **`DataGrid`** to a data source.
- Method **`Clear`** of class **`DataSet`** is called to empty the **`DataSet`** of any prior data.
- The instance properties **`InsertCommand`** and **`UpdateCommand`** of class **`OleDbDataAdapter`** are instances of class **`OleDbCommand`**.

- Property **`CommandText`** of class **`OleDbCommand`** is the **`string`** representing the SQL statement to be executed.
- Method **`ExecuteNonQuery`** of class **`OleDbCommand`** is called to perform the action specified by **`CommandText`** on the database.
- A powerful feature of ADO .NET is its ability to convert data stored in a datasource to XML, and vice versa.
- Method **`WriteXml`** of class **`DataSet`** writes the XML representation of the **`DataSet`** instance to the first argument passed to it. This method has several overloaded versions that allow programmers to specify an output source and a character encoding for the data.
- Method **`ReadXml`** of class **`DataSet`** reads the XML representation of the first argument passed to it into its own **`DataSet`**. This method has several overloaded versions that allow programmers to specify an input source and a character encoding for the data.

TERMINOLOGY

`*` SQL wildcard character
`?` SQL wildcard character
`AcceptChanges` method of **`DataRow`**
`AcceptChanges` method of **`DataTable`**
ADO.NET
`AND`
`ASC` (ascending order)
ascending order (ASC)
asterisk (**`*`**)
atomic operation
`authorISBN` table of **`books`** database
`authors` table of **`books`** database
`books` database
`books` database table relationships
cache
`Clear` method of **`DataSet`**
column
column number
`CommandText` method of **`OleDbCommand`**
commit a transaction
connect to a database
data provider
database
database management system (DBMS)
database table
`DataGrid` class
`DataSet` class
default sorting order is ascending
`DELETE FROM`
`DELETE` statement
`DeleteCommand` property of **`OleDbAdapter`**
`DESC`
disconnected
distributed computing system
`ExecuteNonQuery` method of **`OleDbCommand`**
`ExecuteReader` method of **`OleDbCommand`**
`ExecuteScalar` method of **`OleDbCommand`**
field
`Fill` method of **`OleDbAdapter`**
foreign key
`FROM`
fully qualified name
`GetXml` method of **`DataSet`**
`GROUP BY`
infinity symbol
`INNER JOIN`
`INSERT INTO`
`INSERT` statement
`InsertCommand` property of **`OleDbAdapter`**
interface
joining tables
`LIKE`
many-to-many relationship
match the selection criteria
merge records from Tables
`OLE DB` data provider
`OleDbCommand` class
`OleDbConnection` class
`OleDbDataAdapter` class
one-to-many relationship
`ORDER BY`
ordered
ordering of records
pattern matching
primary key

`Publishers` table of `books` database
query
query a database
`ReadXml` method of `DataSet`
record
record set
`RejectChanges` method of `DataRow`
`RejectChanges` method of `DataTable`
relational database
relational database model
relational database table
result set
roll back a transaction
row
rows to be retrieved
Rule of Entity Integrity
Rule of Referential Integrity
`SELECT`
select
select all fields from a table
`SelectCommand` property of `OleDbAdapter`
selecting data from a table
selection criteria
`SET`
`SetDataBinding` method of `DataGrid`
single-quote character
SQL (Structured Query Language)
SQL keyword
`SQL Server` data provider
SQL statement
square brackets in a query
`System.Data` namespace
`System.Data.OleDb` namespace
`System.Data.SqlClient` namespace
table
table column
table in which record will be updated
table row
`titles` table of `books` database
transaction
transaction processing
`UPDATE`
`Update` method of `OleDbDataAdapter`
`UpdateCommand` property of `OleDbAdapter`
`VALUES`
`WHERE`
`WriteXml` method of `DataSet`

SELF-REVIEW EXERCISES

19.1 Fill in the blanks in each of the following statements:

a) The most popular database query language is __________.
b) A table in a database consists of __________ and __________.
c) Databases can be manipulated in C# as __________ objects.
d) Class __________ enables programmers to display data in `DataSet`s graphically.
e) SQL keyword __________ is followed by selection criteria that specify the records to select in a query.
f) SQL keyword __________ specifies the order in which records are sorted in a query.
g) Selecting data from multiple database tables is called __________ the data.
h) A(n) __________ is/are an integrated collection of data that is/are centrally controlled.
i) A(n) __________ is/are a field(s) in a table for which every entry has/have a unique value in another table and where the field(s) in the other table is/are the primary key for that table.
j) Namespace __________ contains special classes and interfaces for manipulating SQLServer databases in C#.
k) C# uses __________ to transmit data between datasources.
l) Namespace __________ is C#'s general interfacing to a database.

19.2 State which of the following are *true* or *false*. If *false*, explain why.

a) In general, ADO .NET is a disconnected model.
b) SQL can implicitly convert fields with the same name from two or mores tables to the appropriate field.
c) Only the `UPDATE` SQL statement can commit changes to a database.

d) Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity.
e) The **`VALUES`** keyword in an **`INSERT`** statement inserts multiple records in a table.
f) **`SELECT`** statements can merge data from multiple tables.
g) The **`DELETE`** statment deletes only one record in a table.
h) An **`OleDbDataAdapter`** can **`Fill`** a **`DataSet`**.
i) Class **`DataSet`** of namespace **`System.Data`** provides methods that enable developers to create XML documents from datasources.
j) SQLServer is an example of a managed provider.
k) Because C# uses a disconnected model, **`OleDbConnection`**s are optional.
l) It is always faster to assign a value to a variable than to instantiate a new **`object`**.

ANSWERS TO SELF-REVIEW EXERCISES

19.1 a) SQL. b) rows, columns. c) **`DataSet`**. d) **`DataGrid`**. e) **`WHERE`**. f) **`ORDER BY`**. g) joining. h) database. i) foreign key. j) **`System.Data.SqlClient`**. k) XML. l) **`System.Data.OleDb`**.

19.2 a) True. b) False. In a query, failure to provide fully qualified names for fields with the same name in two or more tables is an error. c) False. **`INSERT`** and **`DELETE`** change the database, as well. Do not confuse the SQL **`Update`** statement with method **`OleDbDataAdapter.Update`**. d) True. e) False. An **`INSERT`** statement inserts one record in the table. The **`VALUES`** keyword specifies the comma-separated list of values from which the record is formed. f) True. g) False. The **`DELETE`** statement deletes all records matching its **`WHERE`** clause. h) True. i) True. j) True. k) False. This class is required to connect to a database. l) True.

EXERCISES

19.3 Using the techniques shown in this chapter, define a complete query application for the **`Authors.mdb`** database. Provide a series of predefined queries with an appropriate name for each query displayed in a **`System.Windows.Forms.ComboBox`**. Also, allow users to supply their own queries and add them to the **`ComboBox`**. Provide any queries you feel are appropriate.

19.4 Using the techniques shown in this chapter, define a complete query application for the **`Books.mdb`** database. Provide a series of predefined queries with an appropriate name for each query displayed in a **`System.Windows.Forms.ComboBox`**. Also, allow users to supply their own queries and add them to the **`ComboBox`**. Provide the following predefined queries:
a) Select all authors from the **`Authors`** table.
b) Select all publishers from the **`Publishers`** table.
c) Select a specific author and list all books for that author. Include the title, year and ISBN number. Order the information alphabetically by title.
d) Select a specific publisher and list all books published by that publisher. Include the title, year and ISBN number. Order the information alphabetically by title.
e) Provide any other queries you feel are appropriate.

19.5 Modify Exercise 19.4 to define a complete database-manipulation application for the **`Books.mdb`** database. In addition to the querying capabilities, application should allow users to edit existing data and add new data to the database. Allow the user to edit the database in the following ways:
a) Add a new author.
b) Edit the existing information for an author.
c) Add a new title for an author (remember that the book must have an entry in the **`AuthorISBN`** table). Be sure to specify the publisher of the title.
d) Add a new publisher.
e) Edit the existing information for a publisher.

For each of the preceding database manipulations, design an appropriate GUI to allow the user to perform the data manipulation.

19.6 Modify the address-book example of Fig. 19.29 to enable each address-book entry to contain multiple addresses, phone numbers and e-mail addresses. The user should be able to view multiple addresses, phone numbers and e-mail addresses. [*Note:* This is a large exercise that requires substantial modifications to the original classes in the address-book example.]

19.7 Create an application that allows the user to modify all fields of a database using a transaction process model. The user should be able to find, modify and create entries. The GUI should include buttons **Accept Changes** and **Reject Changes**. Modifications to the datasource should be made when the user clicks **Accept Changes**, by the invoking of method **`Update`** of the **`OleDbDataAdapter`** object. The **`DataSet`**'s **`AcceptChanges`** method should be invoked *after* changes are made to the datasource.

19.8 Write a program that allows the user to modify a database graphically through an XML text editor. The GUI should be able to display the contents of the database and commit any changes in the XML text to the database.

20

Web Forms and ASP.NET

Objectives

- To become familiar with ASP.NET Web Forms.
- To be able to create ASP.NET Web Forms.
- To be able to create a series of Web Forms that work together.
- To be able to use file processing and database techniques with Web Forms.

Things are always at their best in their beginning.
Blaise Pascal

High thoughts must have high language.
Aristophanes

Our life is frittered away by detail … Simplify, simplify.
Henry Thoreau

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

Outline

20.1 Introduction

20.2 How ASP.NET Pages Work

20.3 A Simple HTTP Transaction

20.4 ASP.NET Object Model and WebForms

20.5 Web Controls Part 1: Text and Graphics Controls

20.6 Web Controls Part 2: Web Design Controls and Validation Controls

  20.6.1 Case Study: An Online Guestbook

  20.6.2 Case Study: An Instant Page Content Builder

20.7 Other Topics in ASP.NET: Tracing and Cookieless Session State

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

20.1 Introduction

This chapter introduces *ASP.NET* and discusses its place in C#. ASP.NET is a technology that allows for a simpler and quicker way to create Web applications. This chapter discusses some of the concepts and techniques behind the design and implementation of ASP.NET in the context of Web Forms.

We discuss how these Web Forms also called ASP.NET pages are processed in response to a client (e.g., browser) request. These pages are processed by an *ActiveX component* (i.e., a server-side ActiveX control) called a *scripting engine*. An ASPX file has the file extension **`.aspx`** and contains HTML tags and scripting code.

We present ASP.NET's object model and discuss the structure behind ASP.NET pages. We introduce the various Web controls that are at your disposal for creating functional and practical Web projects.

We present two case studies which combine building ASP.NET pages with other concepts such as file processing and database manipulation. The Instant Page Content Builder Case Study in Section 20.6.2 presents a moderately complicated application consisting of four ASP.NET pages and taking advantage of the database concepts from Chapter 19.

20.2 How ASP.NET Pages Work

The ASP.NET in this chapter demonstrates communication between clients and servers via the HTTP protocol of the World Wide Web. When a server receives a client's HTTP request, the server loads the document (or page) requested by the client. HTML documents are *static documents*—all clients see the same content when requesting an HTML document. ASP.NET is a Microsoft technology for sending to the client dynamic Web content—this includes HTML, Dynamic HTML, ActiveX controls and client-side scripts. The ASP.NET page processes the request (which often includes interacting with a database), and returns the results to the client—normally in the form of an HTML document, but other data formats (e.g., images, binary data, etc.) can be returned.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

The two most common *HTTP request types* (also known as request methods) are **GET** and **POST**. These requests are frequently used to send client form data to a Web server. Although **GET** and **POST** both send information to the server, the method of sending the information is different. A **GET** request sends content as part of the URL (e.g., **www.searchsomething.com/search?query=userquery**). A **POST** request posts form contents to the end of an HTTP request. An HTTP request contains information about the server, client, connection, authorization, etc.

Software Engineering Observation 20.1

The data sent in a **POST** *request is not part of the URL and cannot be seen by the user. Forms that contain many fields are most often submitted by a* **POST** *request. sensitive form fields such as passwords are usually sent using this request type.*

An HTTP request is often used to post data to a server-side form handler that processes the data. For example, when the user responds to a Web-based survey, a request sends the Web server the information specified in the HTML form.

Browsers often *cache* (save on disk) Web pages for quick reloading. This speeds up the user's browsing experience by reducing the amount of data downloaded to view a Web page. Browsers typically do not cache a server's response to a **POST** request because the next **POST** request may not contain the same information. For example several users might request the same Web page to participate in a survey. Each user's response changes the overall results of the survey.

When a Web-based search engine is used, a **GET** request normally supplies the search engine with the information specified in the HTML form. The search engine then performs the search and returns the results as a Web page. These pages are often cached in the event that the user performs the same search again.

Portability Tip 20.1

Because browsers are capable of rendering HTML, an ASP.NET page that generates pure HTML can be rendered on any client browser—regardless of the fact that the page requested ends in **.aspx**.

Software Engineering Observation 20.2

To take advantage of ASP.NET technology, a Web server must have the .NET framework installed.

ASP.NET is a derivation from the *Active Server Page* (ASP) technology with some significant changes and additions. ASP.NET is part of the .NET Framework and enjoys many of the benefits it provides such as garbage collection, integration of components written in different languages and exception handling. Whereas an Active Server Page consists simply of a text-file with HTML and scripts an ASP.NET page of two parts: an ASP.NET page and its code-behind file. The code-behind file contains the code necessary for the page. The code-behind file is compiled. When the page is loaded, the code in the code-behind file is executed—we will discuss this in more detail in Section 20.4.

Both *Web Applications* and *Web Services* (Chapter 21) contain a file with a ***.vsdisco*** extension generated by Visual Studio. This is the *discovery* file for the project that stores information about the locations of various resources required for the project. Despite it's possibly misleading **.vsdisco** extension it is an XML file and allows developers to programmatically discover the Web Service.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

20.3 A Simple HTTP Transaction

Before exploring how ASP.NET operates, it is necessary to have a basic understanding of networking and how the World Wide Web works. In this section, we will examine the inner workings of the *HyperText Transfer Protocol* (*HTTP*) and discuss what goes on behind the scenes when a browser displays a Web page. HTTP describes a set of *methods* and *headers* that allows clients and servers to interact and exchange information in a uniform and predictable way.

A Web page in its simplest form is nothing more than an *HTML* (*HyperText Markup Language*) document. This document is just a plain text file containing markings (*markup* or *tags*) that describe to a Web browser how to display and format the information in the document. For example, the HTML

```
<title>My Web Page</title>
```

indicates to the browser that the text between the opening **`<title>`** tag and the closing **`</title>`** tag is the title of the Web page. HTML documents can also contain *hypertext* information (usually called *hyperlinks*) that create links to different pages or to other portions of the same page. When the user activates a hyperlink (usually by clicking on it with a mouse), a new Web page (or a different part of the same Web page) is loaded for the user to view. Note that in HTML, tags are not case sensitive, so **`<TITLE>`** works the same as **`<title>`**.

Any HTML file available for viewing over the Web has a *URL* (*Universal Resource Locator*) associated with it—an address of sorts. The URL contains information that directs a browser to the resource (most often a Web page) that the user wishes to access. For example, let us break down the URL

```
http://www.deitel.com/books/downloads.htm
```

into its basic components. The **`http://`** indicates that the resource is to be obtained using the Hypertext Transfer Protocol.

The middle portion, **`www.deitel.com`**, is the *hostname* of the server. The hostname is the name of the computer where the resource resides, and likewise, this computer is usually referred to as the *host*, because it houses and maintains the resource. The hostname **`www.deitel.com`** is translated into an *IP address* (**`207.60.134.230`**) that identifies the server (just as a telephone number uniquely defines a particular phone line). The translation of the hostname into an IP address is normally performed by a *domain name server* (*DNS*), a computer that maintains a database of hostnames and their corresponding IP addresses. Many people refer to this translation operation as a *DNS lookup*.

The name of the resource being requested, **`/books/downloads.htm`** (an HTML document), is the remainder of the URL. This portion of the URL specifies both the name of the resource (**`downloads.htm`**) and its path (**`/books`**). The path could represent an actual directory in the Web server's file system. However, for security reasons, the path often is a *virtual directory*. In this case, the server translates the path into a real location on the server (or even on another computer), thus hiding the true location of the resource. In fact, it is even possible that the resource is created dynamically and does not reside anywhere on the server computer. Thus, the URL uses the hostname to locate the correct server, and the server uses the path and resource information to locate (or create) the

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

resource to respond to the client's request. As we will see, URLs can also be used to provide input to a program on the server.

Now we consider how a browser, when given a URL, performs a simple HTTP transaction to fetch and display a Web page. Figure 20.1 illustrates the transaction in detail. The transaction is performed between a Web browser application on the client side and a Web server application on the server side.

In step 1 of Fig. 20.1, the browser sends an HTTP request message to the server. The request (in its simplest form) looks something like the following:

```
GET /books/downloads.htm HTTP/1.0
```

The word ***GET*** is an *HTTP method* (a term for functions in HTTP) indicating that the client wishes to get a resource. The remainder of the request provides the name and path of the resource (an HTML document) and the protocol's name and version number (**`HTTP/1.0`**).

Any server that understands HTTP (version 1.0) will be able to translate this request and respond appropriately. Part 2 of Fig. 20.1 shows the results of a successful request. The server first responds with a line indicating the HTTP version, followed by a numeric code and a phrase describing the status of the transaction. For example,

```
HTTP/1.0 200 OK
```

indicates success, while

```
HTTP/1.0 404 Not found
```

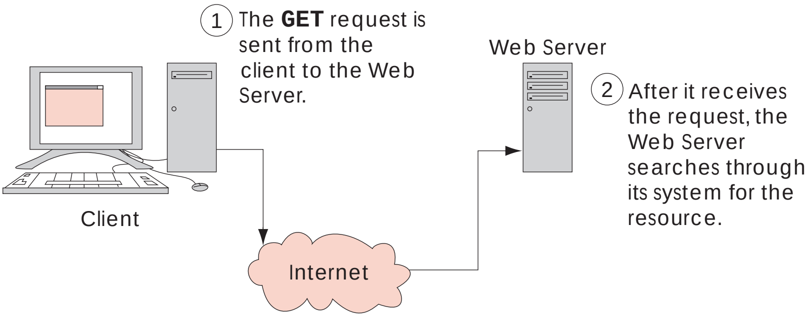

① The **GET** request is sent from the client to the Web Server.
Web Server
② After it receives the request, the Web Server searches through its system for the resource.
Client
Internet

Fig. 20.1 Client interacting with server and Web server. Step 1: The **`GET`** request, **`GET /books/downloads.htm HTTP/1.0`** (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

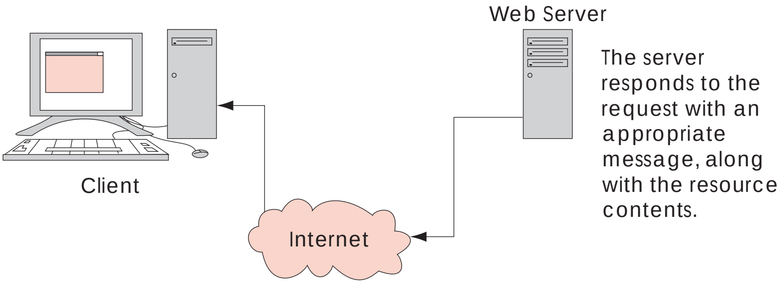

Web Server
The server responds to the request with an appropriate message, along with the resource contents.
Client
Internet

Fig. 20.1 Client interacting with server and Web server. Step 2: The HTTP response, HTTP/1.0 200 OK (part 2 of 2).

informs the client that the requested resource was not found on the server in the specified location.

The server normally then sends one or more *HTTP headers*, which provide additional information about the data being sent. In this case, the server is sending an HTML text document, so the HTTP header reads

```
Content-type: text/html
```

This information is known as the *MIME type* of the content. Each type of data sent from the server has a MIME type that helps the browser determine how to process the data it receives. For example, the MIME type **`text/txt`** indicates that the data are text that should be displayed without attempting to interpret any of the content as HTML markup. Similarly, the MIME type **`image/gif`** indicates that the content is a GIF image. When this MIME type is received by the browser, it attempts to display the image.

The header or set of headers is followed by a blank line, which indicates to the client that the server is finished sending HTTP headers. The server then sends the text in the requested HTML document (**`downloads.htm`**). The connection is terminated when the transfer of the resource is complete. The client-side browser interprets the HTML it receives and displays (or renders) the results.

Common Programming Error 20.1

Forgetting to place a blank line after a header is a logic error.

20.4 ASP.NET Object Model and WebForms

To run and create ASP.NET a Web server needs to be installed. We use the IIS Web Server in all our examples, since IIS is already required for Visual Studio.NET to be installed. If you have problems installing IIS go the **Downloads/Resources** link on the Deitel Web site at **`www.deitel.com`**.

To create an ASP.NET Web Form project create a project of type **ASP.NET Web Application**. By default this creates a project in the **`C:\inetpub\wwwroot`** root directory of IIS. The solution file for the project (the **`.sln`** file) is placed in a folder with the project name in the default directory for all Visual Studio.NET projects—in the **Visual**

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

Studio Projects folder in your **My Documents** folder. Opening this file will open up your ASP.NET project.

A Web Form is a Web version of Windows Form. It will always have the extension **.aspx** and is there to contain all the visual components of your page. To design the Web Form right click on it in the **Solution Explorer** and select **View Designer**.

Designing a Web Form is as simple as designing a Windows Form. To add controls such as buttons or textboxes to the page they can be dragged from the **Toolbox** to the Web Form. All controls are objects, including the page itself. This means they may have properties, methods and events. The properties and events can be changed and set just like before in the **Properties** window. These controls, however, are contained in the ***System.Web.UI.WebControls*** namespace, not in the **System.Windows.Forms** namespace. Even though some classes such as the **TextBox** class belong in both namespaces, they are in fact two completely different classes. One is designed to be used on Windows Forms, the other on ASP.NET pages.

Common Programming Error 20.2

It is easy to confuse the classes from ***System.Windows.Forms*** *and* ***System.Web.UI.WebControls*** *and expect them to be identical. They do share some similar properties and methods, but not others.*

By default the ***pageLayout*** property of the page will be in ***GridLayout*** mode by default, meaning all controls will be placed on a grid and assigned x and y coordinates—their placement does not depend on one another. This can be changed by changing the **pageLayout** property of ***Document*** to ***FlowLayout***—all controls are sequentially lined up as soon as they are placed. **Document** is the name Visual Studio.NET gives the page you are currently working on. We use **GridLayout** mode for our examples.

Every ASP.NET page created in Visual Studio will have a corresponding C# class which can contain event handlers, initialization code and anything else that the page may need. The file in which this class is contained is called the code-behind file. To view the contents of this C# class right click on the ASPX file in the **Solution Explorer** and choose **View Code**.

Every ASP.NET page is actually an object of type **Page** in namespace ***System.Web.UI***. Among others, **Page** has three properties: ***Request*** of type ***HttpRequest***, ***Response*** of type ***HttpResponse*** and ***Session*** of type ***HttpSessionState***.

Request provides information about the incoming HTTP request such as the values of the variables in our form. So if this page is being loaded in response to a user clicking a button after filling out some fields (textboxes, choosing from a drop–down list, etc.) on a page, the **Request** object allows us to retrieve those values. **Response** sends information back to the clients browser. For example, it can be used to send HTML code or redirect the user to a different page. **Session** provides information about the current session. It stores *session variables*—variables that can be accessed by any page during the same session. We will use these classes extensively in the case study of Section 20.6.2.

Now let us look at Fig. 20.2 for a simple Web Form example which updates the time every minute.

```
<%-- Fig. 20.2 ---%>
<%-- A page that updates the time every minute ---%>

<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
    AutoEventWireup="false" Inherits="WebTime.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta name="GENERATOR"
            Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript"
            content="JavaScript (ECMAScript)">
      <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
      <meta http-equiv="REFRESH" Content="60;
            URL=WebForm1.aspx">
   </HEAD>
   <body MS_POSITIONING="GridLayout">
      <form id="Form1" method="post" runat="server">
         <asp:Label id="Label1" style="Z-INDEX: 101;
            LEFT: 23px; POSITION: absolute; TOP: 66px"
            runat="server" Width="229px" Height="55px"
            ForeColor="#00C000" BackColor="Black"
            Font-Size="XX-Large">Label
         </asp:Label>
         <asp:Label id="Label2" style="Z-INDEX: 102;
            LEFT: 28px; POSITION: absolute; TOP: 31px"
            runat="server" Width="221px" Height="19px"
            Font-Size="Medium">A Simple ASP.NET Example
         </asp:Label>
      </form>
   </body>
</HTML>
```

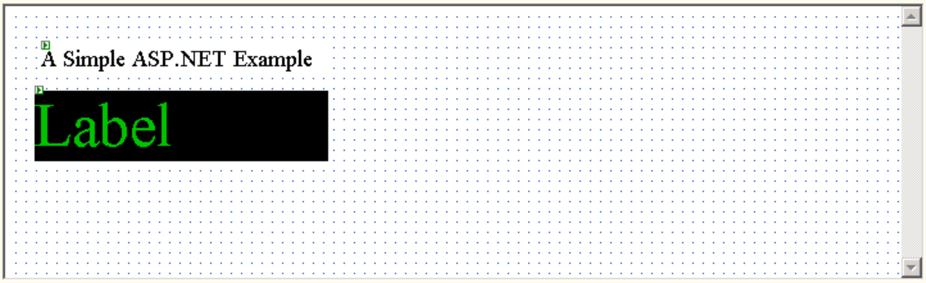


Fig. 20.2 The ASPX code for a page that updates the time every minute.

Almost all of the code you see here is generated for you by Visual Studio when you drag two **`Label`**s onto the form and set their properties. The screen capture in Fig. 20.2 shows what the page looks like at design time. The code in Fig. 20.2 and other ASPX listings in this chapter have been reformatted for display purposes. Let us take a look at the generated code in detail.

Lines 4–5 use *processing directives* to specify some of the attributes of the page. The language of the page is declared to be C#, the code-behind file is **`WebForm1.aspx.cs`**. The "**`<%`**" and "**`%>`**" tags indicate that whatever is in between these constitutes a *client-side script*. A script is a set of instructions for another program, not a CPU, to execute. A client-side script is a script that executes on the clients computer. The output is displayed on their Web browser.

The ***`AutoEventWireup`*** attribute is set to **`false`**; when it is set to **`true`** any event handler that one writes in a script will be attached to the page provided it has the proper naming convention. For example, the ***`Init`*** event occurs during the initialization stage of the page, the name convention would be to name the event handler for it **`Page_Init`**. The ***`Inherits`*** attribute specifies what class this ASP.NET class inherits from—in this case its **`WebForm1`**. **`WebForm1`** does not inherit from it in the way we have talked about so far because the ASP.NET page is HTML code and not a C# class.

Common Programming Error 20.3

It is common for programmers to rename their Web Forms. Doing this in Visual Studio means that the programmer must manually change the value of the **`Inherits`** *attribute. A way to avoid this problem is to delete the other Web Form and create a new once with the desired name.*

The only code we manually placed consists of lines 16–17, which tell the page to refresh every 60 seconds by going to **`WebForm1.aspx`**—itself.

Lines 20–32 define the **`Form`** which will contain all our controls—two **`Label`**s. Lines 21–26 and 27–31 show how the two **`Label`**s are created via HTML. All the properties that we set in the **Properties** window such as **`Font`** and **`Text`** are shown as attributes here.

Below, in Fig. 20.3 we see the corresponding code-behind file, **`WebForm1.aspx.cs`**. Note that the project must be compiled before the page can be displayed.

```
// Fig. 20.3
// The code-behind file for a page that updates the current
// time every minute

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
```

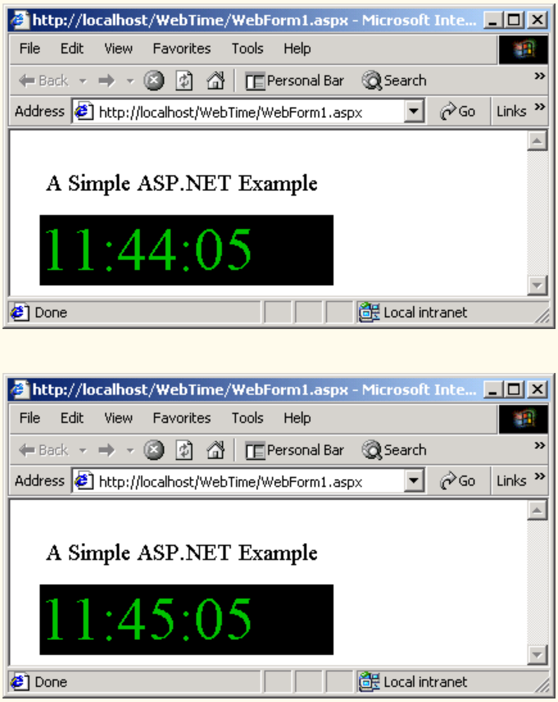
Fig. 20.3 Code-behind file for a page that updates the time every minute (part 1 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
using System.Web.UI.HtmlControls;

namespace WebTime
{
   /// <summary>
   /// Summary description for WebForm1.
   /// </summary>
   public class WebForm1 : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Label Label1;
      protected System.Web.UI.WebControls.Label Label2;

      public WebForm1()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      private void Page_Load( object sender, System.EventArgs e )
      {
         // Put user code to initialize the page here
      }

      private void Page_Init( object sender, EventArgs e )
      {
         //
         // CODEGEN: This call is required by the ASP.NET Web Form
         // Designer.
         //
         InitializeComponent();

         Label1.Text =
            string.Format( "{0:D2}", DateTime.Now.Hour) + ":" +
            string.Format( "{0:D2}", DateTime.Now.Minute ) +
            ":" + string.Format( "{0:D2}", DateTime.Now.Second );
       }

       #region Web Form Designer generated code
       /// <summary>
       /// Required method for Designer support - do not modify
       /// the contents of this method with the code editor.
       /// </summary>
       private void InitializeComponent()
       {
          this.Load += new System.EventHandler( this.Page_Load );
       }
       #endregion
   }
}
```

Fig. 20.3 Code-behind file for a page that updates the time every minute (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01



http://localhost/WebTime/WebForm1.aspx - Microsoft Inte...
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Address
http://localhost/WebTime/WebForm1.aspx
Go
Links
Done
Local intranet

A Simple ASP.NET Example

11:44:05

A Simple ASP.NET Example

11:45:05

Fig. 20.3 Code-behind file for a page that updates the time every minute (part 3 of 3).

For the purposes of this first example we have left all the Visual Studio generated code in place. As you see when the object for the page is created an event handler is created and attached for the `Init` event. This event happens during the initialization of the page and is then fired. This event first calls `InitializeComponent`—much like with Windows Forms this method is required for designer support. `InitializeComponent` is used to create event handlers for controls on the page and for initialization purposes such as creating a database connection. `InitializeComponent` creates and attaches an event handler for the `Load` event. This event fires when the page is loaded. After `InitializeComponent` lines 44–47 execute, setting `Label1` to contain the current time. The `Init` event always fires before the `Load` event.

After you create your Web Form you may view and test it in three different ways. You may select **Start** from the **Debug** menu which will run the application by opening up a new browser. Closing this browser will terminate the application.

You may also right-click either on the ASPX file from the **Solution Explorer** or the Web Form designer and select **View In Browser**. This will open up a browser within Visual Studio and will allow you to preview the page—this is usually faster then running the project.

A third way is to open up a browser yourself and type in the Web address of your project. When testing a project on the same computer you should type "`http://localhost/`*`ProjectFolder`*`/`*`PageName`*`.aspx`" where `ProjectFolder` is the folder

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

where your page resides (usually the name of your project) within **`C:/inetpub/wwwroot`** and **`PageName`** is the name of your ASP.NET page.

20.5 Web Controls Part 1: Text and Graphics Controls

This section discusses some of the features that are at your disposal when designing Web Forms. The **Web Form** tab of the **Toolbox** has numerous controls that are helpful when designing them.

Web pages often contain ads by corporate sponsors advertising their service or products. They may only have one or two spots on their page for ads, and want to be able to cycle through all the different ones they have easily. The ***`AdRotator`*** class is designed for such a purpose. Using the information in an XML file, the **`AdRotator`** control can randomly choose an image to display, set it to link to the appropriate page and show the appropriate caption. If the browser does not support images it will show the alternate text as directed by the XML file. Figure 20.4 demonstrates how to use the **`AdRotator`** class.

```
<%-- Fig. 20.4 ---%>
<%-- A Web Form that demonstrates class AdRotator ---%>

<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
    AutoEventWireup="false"
    Inherits="AdRotatorTest.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta name="GENERATOR"
            Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript"
            content="JavaScript (ECMAScript)">
      <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
   </HEAD>
   <body MS_POSITIONING="GridLayout">
      <form id="Form1" method="post" runat="server">
         <asp:AdRotator id="AdRotator1" style="Z-INDEX: 101;
            LEFT: 43px; POSITION: absolute; TOP: 73px"
            runat="server" Width="108px" Height="72px"
            AdvertisementFile="AdRotatorInformation.xml">
         </asp:AdRotator>
         <asp:Label id="Label1" style="Z-INDEX: 102;
            LEFT: 42px; POSITION: absolute; TOP: 39px"
            runat="server"
            Font-Size="Medium">AdRotator Example
         </asp:Label>
      </form>
   </body>
</HTML>
```

Fig. 20.4 A Web Form that demonstrates the **`AdRotator`** class .

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

The above ASP.NET page is similar to the previous example. Instead of having two **`Label`**s, this page has a **`Label`** and an **`AdRotator`** called **`AdRotator1`**. **`AdRotator1`**'s ***`AdvertisementFile`*** property has been set to **`AdRotatorInfo.xml`** (line 23). Notice that we set this property—as we do in many cases—in the **Properties** window in Visual Studio, but this gets reflected in the generated code. This is the file **`AdRotator1`** will look in to get all the information it needs about which ad it will show next. As you can see in Fig. 20.5, no additional code needed to be placed in the code-behind file (Fig. 20.5).

```
// Fig. 20.5
// The code-behind file for a page that demonstrates
// the AdRotator class.

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace AdRotatorTest
{
   public class WebForm1 : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.AdRotator AdRotator1;
      protected System.Web.UI.WebControls.Label Label1;

      public WebForm1()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      private void Page_Load( object sender,
         System.EventArgs e )
      {
      }

      private void Page_Init( object sender, EventArgs e )
      {
         InitializeComponent();
      }

      // Visual Studio generated code
   }
}
```

Fig. 20.5 Code-behind file for page demonstrating the **`AdRotator`** class (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

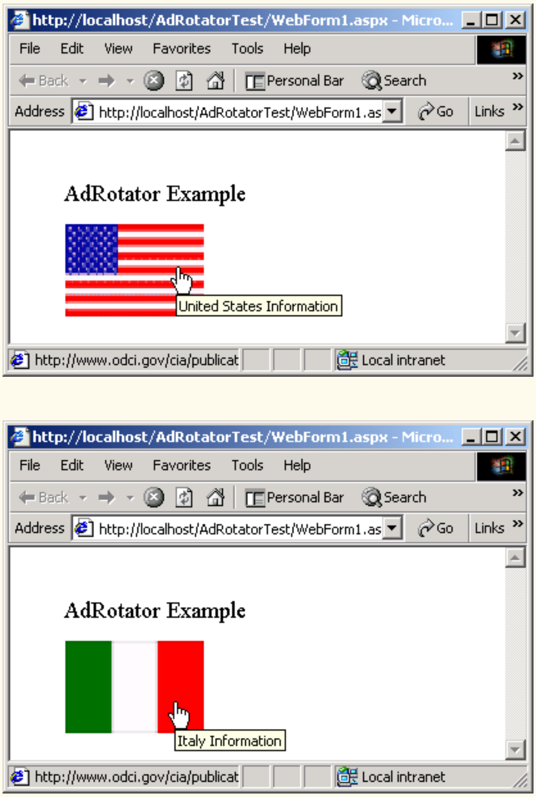

http://localhost/AdRotatorTest/WebForm1.aspx - Micro...
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Address
http://localhost/AdRotatorTest/WebForm1.as
Go
Links
AdRotator Example
United States Information
http://www.odci.gov/cia/publicat
Local intranet
http://localhost/AdRotatorTest/WebForm1.aspx - Micro...
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Address
http://localhost/AdRotatorTest/WebForm1.as
Go
Links
AdRotator Example
Italy Information
http://www.odci.gov/cia/publicat
Local intranet

Fig. 20.5 Code-behind file for page demonstrating the **`AdRotator`** class (part 2 of 2).

`AdRotatorInformation.xml` (Fig. 20.6) contains several ***`Ad`*** nodes, each of which contain a complete set of information about each ad. The ***`ImageUrl`*** tag specifies the location of the image that will be displayed when this ad is chosen. The ***`NavigateUrl`*** tag specifies the URL of the site users will be sent to when they on the ad. The ***`AlternateText`*** tag declares the text that should be displayed in case the browser in case it does not support images. It is also the caption that will show in Internet Explorer when the mouse is over the image as can be see in the screen shots in Fig. 20.5. The ***`Impressions`*** tag specifies the frequency with which this image should be displayed. An ad having a higher **`Impressions`** value will be displayed more often then one with a lower value. In our example all the ads will be displayed with equal frequency because **`Impressions`** is always 1.

```
<%-- Fig. 20.6 --%>
<%-- XML file containing ad information --%>

<?xml version="1.0" encoding="utf-8"?>
```

Fig. 20.6 XML file containing **`AdRotator`** information.

```
<Advertisements>
  <Ad>
    <ImageUrl>images/us.gif</ImageUrl>
    <NavigateUrl>
    http://www.odci.gov/cia/publications/factbook/geos/us.html
    </NavigateUrl>
    <AlternateText>United States Information</AlternateText>
    <Impressions>1</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>images/france.gif</ImageUrl>
    <NavigateUrl>
    http://www.odci.gov/cia/publications/factbook/geos/fr.html
    </NavigateUrl>
    <AlternateText>France Information</AlternateText>
    <Impressions>1</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>images/germany.gif</ImageUrl>
    <NavigateUrl>
    http://www.odci.gov/cia/publications/factbook/geos/gm.html
    </NavigateUrl>
    <AlternateText>Germany Information</AlternateText>
    <Impressions>1</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>images/italy.gif</ImageUrl>
    <NavigateUrl>
    http://www.odci.gov/cia/publications/factbook/geos/it.html
    </NavigateUrl>
    <AlternateText>Italy Information</AlternateText>
    <Impressions>1</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>images/spain.gif</ImageUrl>
    <NavigateUrl>
    http://www.odci.gov/cia/publications/factbook/geos/sp.html
    </NavigateUrl>
    <AlternateText>Spain Information</AlternateText>
    <Impressions>1</Impressions>
  </Ad>
</Advertisements>
```

Fig. 20.6 XML file containing **`AdRotator`** information.

20.6 Web Controls Part 2: Web Design Controls and Validation Controls

Web controls placed on a page are there to provide some sort of functionality—allowing input, verifying input, allowing users to interact with the page. Some controls provided with ASP.NET are: ***`TextBox`***, ***`HyperLink`***, ***`Button`***, ***`LinkButton`***, ***`CheckBox`***, ***`CheckBoxList`***, ***`RadioButtonList`***, ***`Table`***, ***`RequiredFieldValidator`***, ***`Compare-`***

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

Validator, ***RangeValidator***, ***RegularExpressionValidator*** and ***CustomValidator***.

The example in Fig. 20.7 is an example of using a **RegularExpressionValidator** and a **RequiredFieldValidator**. A validator is a control which checks that the data in another control is valid and consistent. This is useful when validating information—for example we may need to check that the user filled out every field, or that the zip code field contains 5 digits.

This page accepts a phone number in the form **xxx-xxxx** (where each **x** represents a digit) as input, and gives all the possible words that can be made with the first three letters, and all the possible words that can be made with the last four.

```
<%-- Fig. 20.7 --%>
<%-- A Web Form demonstrating the use of validators --%>

<%@ Page language="c#" Codebehind="Generator.aspx.cs"
    AutoEventWireup="false"
    Inherits="WordGenerator.Generator" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta content="Microsoft Visual Studio 7.0"
            name="GENERATOR">
      <meta content="C#" name="CODE_LANGUAGE">
      <meta content="JavaScript (ECMAScript)"
            name="vs_defaultClientScript">
      <meta
      content="http://schemas.microsoft.com/intellisense/ie5"
      name="vs_targetSchema">
   </HEAD>
   <body MS_POSITIONING="GridLayout">
      <form id="Generator" method="post" runat="server">
         <asp:RegularExpressionValidator
            id="RegularExpressionValidator1"
            style="Z-INDEX: 101; LEFT: 9px;
            POSITION: absolute; TOP: 5px" runat="server"
            ErrorMessage=
            "Please enter a phone number in the form xxx-xxxx"
            ValidationExpression="\d{3}-\d{4}"
            ControlToValidate="phoneTextBox">
         </asp:RegularExpressionValidator>
         <asp:TextBox id="phoneTextBox" style="Z-INDEX: 102;
            LEFT: 12px; POSITION: absolute; TOP: 36px"
            runat="server" Width="106px" Height="24px">
         </asp:TextBox>
         <asp:Button id="Button1" style="Z-INDEX: 103;
            LEFT: 14px; POSITION: absolute; TOP: 77px"
            runat="server" Text="Submit">
         </asp:Button>
         <asp:RequiredFieldValidator
            id="RequiredFieldValidator1" style="Z-INDEX: 104;
```

Fig. 20.7 A Web Form that generates all possible words from a provided phone number (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
40              LEFT: 132px; POSITION: absolute; TOP: 39px"
41              runat="server"
42              ErrorMessage="Please enter a phone number"
43              ControlToValidate="phoneTextBox">
44           </asp:RequiredFieldValidator>
45        </form>
46     </body>
47  </HTML>
```

Fig. 20.7 A Web Form that generates all possible words from a provided phone number (part 2 of 2).

Lines 21–29 create a **RegularExpressionValidator** named **RegularExpressionValidator1**. Line 26 shows that value of the **ErrorMessage** property. This is the message the user will see if the information entered is not validated by **RegularExpressionValidator1**. Line 27 set the regular expression the validator will be using to check its input by setting the ***ValidationExpression*** property. When clicking on the **ValidationExpression** property of **RegularExpressionValidator1** a box pops up containing several common regular expressions such as phone numbers and zip codes. We do not want the user to be able to enter an area code, so we type in a custom regular expression as above. Line 28, is the key line which connects our **TextBox** **phoneTextBox** to **RegularExpressionValidator1** by setting the ***ControlToValidate*** property of **RegularExpressionValidator1**. Now **RegularExpressionValidator1** knows that it will be getting its input from **phoneTextBox** and will validate its text. If the user types nothing and simply clicks the **Button** however, **RegularExpressionValidator** will not catch the problem; to fix this problem we create a second validation control.

Lines 38–44 define a **RequiredFieldValidator** called **RequiredFieldValidator1**. **RequiredFiledValidator**s have similar properties as **RegularExpressionValidator**s. They however merely ensure that the user filled out the control represented by the **ControlToValidate** property of the **RequiredFieldValidator**. This second validator ensures that the user types at least something in the **TextBox**—if the input is not in a valid format **RegularExpressionValidator1** will catch the mistake. The code-behind file for the word generator is shown in Fig. 20.8.

```
// Fig. 20.8
// The code-behind file for a page that generates words
// given a phone number.

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
```

Fig. 20.8 Code-behind file for Word Generator page (part 1 of 5).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace WordGenerator
{
   public class Generator : System.Web.UI.Page
   {
      protected
         System.Web.UI.WebControls.RegularExpressionValidator
         RegularExpressionValidator1;
      protected System.Web.UI.WebControls.TextBox
         phoneTextBox;
      protected
         System.Web.UI.WebControls.RequiredFieldValidator
         RequiredFieldValidator1;
      protected System.Web.UI.WebControls.Button Button1;

      public Generator()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      private void Page_Load( object sender, System.EventArgs e )
      {
         // if this is not the first time the page is loaded
         if ( IsPostBack )
         {
            // retrieve the number and remove the "-"
            string number = Request.Form[ "phoneTextBox" ];
            number = number.Remove( 3, 1 );

            // calculate all the words for the first 3 numbers
            Response.Write( "<br><br><br><br><br><br>Here " +
               "are the words for the first three digits:" +
               "<br>" );
            ComputeWords( number.Substring( 0, 3 ), "" );

            // calculate all the words for the last 4 numbers
            Response.Write( "<br>Here are the words for the"
               + " last four digits:<br>" );
            ComputeWords( number.Substring( 3 ), "" );
         }
      }

      private void Page_Init( object sender, EventArgs e )
      {
         InitializeComponent();
      }

      // Visual Studio generated code

      public void ComputeWords( string number, string sofar )
      {
```

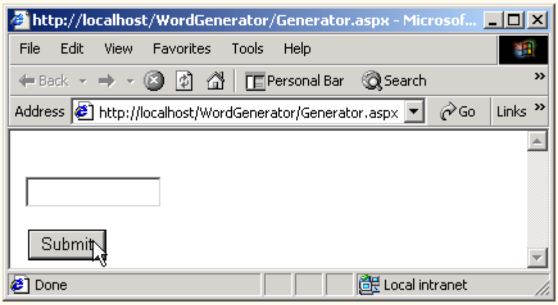
Fig. 20.8 Code-behind file for Word Generator page (part 2 of 5).

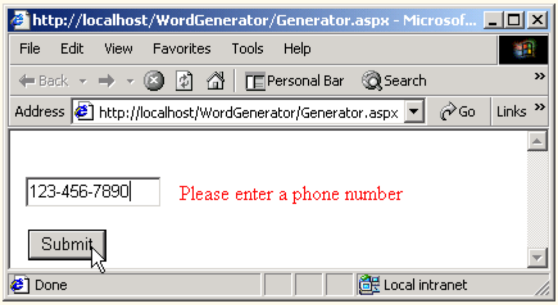
```
            // if number is empty, time to print the word
            if ( number == "" )
            {
               Response.Write( sofar.ToString() + "<br>" );
               return;
            }

            // retrieve the first number from the string
            int cur = Int32.Parse( number.Substring( 0, 1 ) );

            // delete the first number from the string
            number = number.Remove( 0, 1 );

            // depending on the number, call ComputeWords
            // recursively with the 3 (in the case of the number
            // 0 it is 2) possibilities of letters.
            switch ( cur )
            {
               case 0: ComputeWords( number, sofar + "q" );
                  ComputeWords( number, sofar + "z" );
                  break;
               case 1: ComputeWords( number, sofar + " " );
                  break;
               case 2: ComputeWords( number, sofar + "a" );
                  ComputeWords( number, sofar + "b" );
                  ComputeWords( number, sofar + "c" );
                  break;
               case 3: ComputeWords( number, sofar + "d" );
                  ComputeWords( number, sofar + "e" );
                  ComputeWords( number, sofar + "f" );
                  break;
               case 4: ComputeWords( number, sofar + "g" );
                  ComputeWords( number, sofar + "h" );
                  ComputeWords( number, sofar + "i" );
                  break;
               case 5: ComputeWords( number, sofar + "j" );
                  ComputeWords( number, sofar + "k" );
                  ComputeWords( number, sofar + "l" );
                  break;
               case 6: ComputeWords( number, sofar + "m" );
                  ComputeWords( number, sofar + "n" );
                  ComputeWords( number, sofar + "o" );
                  break;
               case 7: ComputeWords( number, sofar + "p" );
                  ComputeWords( number, sofar + "r" );
                  ComputeWords( number, sofar + "s" );
                  break;
               case 8: ComputeWords( number, sofar + "t" );
                  ComputeWords( number, sofar + "u" );
                  ComputeWords( number, sofar + "v" );
                  break;
               case 9: ComputeWords( number, sofar + "w" );
                  ComputeWords( number, sofar + "x" );
                  ComputeWords( number, sofar + "y" );
```

Fig. 20.8 Code-behind file for Word Generator page (part 3 of 5).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

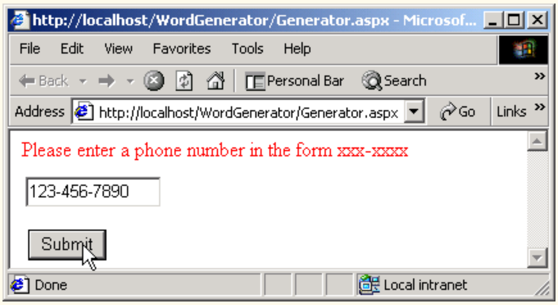
```
               break;
         } // end of switch statement
      } // end of ComputeWords
   }
}
```


http://localhost/WordGenerator/Generator.aspx - Microsof...
File Edit View Favorites Tools Help
Back Personal Bar Search
Address http://localhost/WordGenerator/Generator.aspx Go Links
Submit
Done
Local intranet


http://localhost/WordGenerator/Generator.aspx - Microsof...
File Edit View Favorites Tools Help
Back Personal Bar Search
Address http://localhost/WordGenerator/Generator.aspx Go Links
Submit
Done
Local intranet

123-456-7890

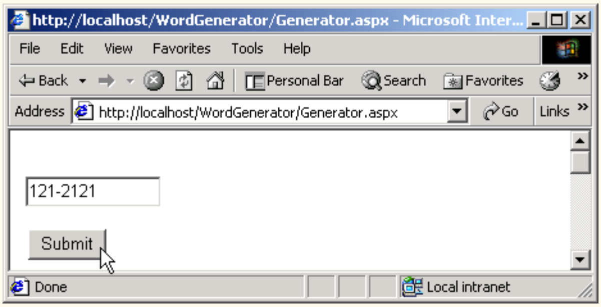
Please enter a phone number

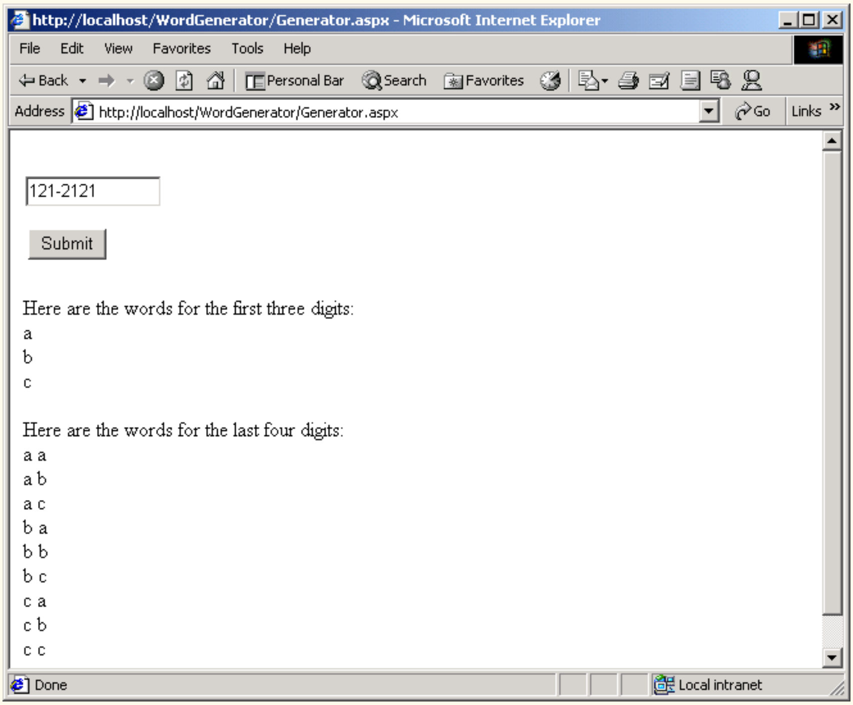

http://localhost/WordGenerator/Generator.aspx - Microsof...
File Edit View Favorites Tools Help
Back Personal Bar Search
Address http://localhost/WordGenerator/Generator.aspx Go Links
Submit
Done
Local intranet

Please enter a phone number in the form xxx-xxxx

123-456-7890

Fig. 20.8 Code-behind file for Word Generator page (part 4 of 5).


http://localhost/WordGenerator/Generator.aspx - Microsoft Inter...
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/WordGenerator/Generator.aspx
Go Links
121-2121
Submit
Done
Local intranet


http://localhost/WordGenerator/Generator.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/WordGenerator/Generator.aspx
Go Links
121-2121
Submit
Done
Local intranet

```
Here are the words for the first three digits:
a
b
c

Here are the words for the last four digits:
a a
a b
a c
b a
b b
b c
c a
c b
c c
```

Fig. 20.8 Code-behind file for Word Generator page (part 5 of 5).

We modified the **`Page_Load`** event handler and added some instructions if the page is being loaded due to a *postback*. We want to retrieve the text in **`phoneTextBox`**, compute all the words for the first three numbers, and compute all the words for the last four. Method **`ComputeWords`** is called with the substring containing the first three numbers and an empty **`string`**. This empty **`string`** represent what word we have computed so far. **`ComputeWords`**, defined on lines 64–122, is a recursive method. If **`number`** is an empty **`string`**, it is time to print out the word, so the contents of **`sofar`** are printed and the function exits. Otherwise the first character is converted into an **`int cur`** and removed from **`number`**. The **`switch`** statement determines which letters could possible be appended to the **`string sofar`** based on **`cur`**. **`ComputeWords`** is then called again two or three times depending on how many letters that number can represent. In each case, **`ComputeWords`** is called again with **`number`**—now with its front character removed—and **`sofar`** concatenated with one of the letters this number can represent. In this manner all possible letter combinations of these numbers are printed out.

20.6.1 Case Study: An Online Guestbook

This section presents Web guestbook application which allows users to leave messages and read those of others. Included with each message is the name of the person who left it and e-mail link to the e-mail address the user provided. Here we put the concept we have used throughout the chapter to develop an application of medium difficulty using ASP.NET and file processing techniques.

Figure 20.9 has the code listing for the ASPX part of an online guestbook.

```
<%-- Fig. 20.9 --%>
<%-- A Web Form demonstrating a guestbook. --%>

<%@ Page language="c#" Codebehind="GuestbookPage.aspx.cs"
    AutoEventWireup="false"
    Inherits="Guestbook.GuestbookPage" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta name="GENERATOR"
            Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript"
            content="JavaScript (ECMAScript)">
      <meta name="vs_targetSchema"
      content="http://schemas.microsoft.com/intellisense/ie5">
   </HEAD>
   <body MS_POSITIONING="GridLayout">
      <form id="Guestbook" method="post" runat="server">
         <asp:Table id="greetingTable" style="Z-INDEX: 101;
            LEFT: 30px; POSITION: absolute; TOP: 17px"
            runat="server">
            <asp:TableRow>
               <asp:TableCell ForeColor="Blue"
                  Font-Size="X-Large" Text=
                  "Please leave a message in our guestbook:">
               </asp:TableCell>
            </asp:TableRow>
         </asp:Table>
         <asp:Table id="messageTable" style="Z-INDEX: 102;
            LEFT: 32px; POSITION: absolute; TOP: 409px"
            runat="server">
         </asp:Table>
         <asp:Label id="Label1" style="Z-INDEX: 103;
            LEFT: 37px; POSITION: absolute; TOP: 118px"
            runat="server" Width="44px"
            Height="19px">Your Name:
```

Fig. 20.9 ASPX listing for the guestbook page (part 1 of 3).

```
         </asp:Label>
         <asp:TextBox id="nameTextBox" style="Z-INDEX: 104;
            LEFT: 98px; POSITION: absolute; TOP: 126px"
            runat="server" Width="447px" Height="24px">
         </asp:TextBox>
         <asp:Label id="Label2" style="Z-INDEX: 105;
            LEFT: 34px; POSITION: absolute; TOP: 172px"
            runat="server" Width="46px"
            height="19px">Your email address:
         </asp:Label>
         <asp:TextBox id="emailTextBox" style="Z-INDEX: 106;
            LEFT: 100px; POSITION: absolute; TOP: 186px"
            runat="server" Width="449px" Height="24px">
         </asp:TextBox>
         <asp:TextBox id="messageTextBox" style="Z-INDEX: 107;
            LEFT: 99px; POSITION: absolute; TOP: 241px"
            runat="server" Width="449px" Height="113px"
            TextMode="MultiLine">
         </asp:TextBox>
         <asp:Label id="Label3" style="Z-INDEX: 108;
            LEFT: 38px; POSITION: absolute; TOP: 263px"
            runat="server" Width="49px"
            Height="19px">Tell the world:
         </asp:Label>
         <asp:Button id="Button1" style="Z-INDEX: 109;
            LEFT: 196px; POSITION: absolute; TOP: 368px"
            runat="server" Text="Submit">
         </asp:Button>
         <asp:Button id="Button2" style="Z-INDEX: 110;
            LEFT: 357px; POSITION: absolute; TOP: 368px"
            runat="server" Text="Clear" Height="24px"
            Width="61px">
         </asp:Button>
      </form>
   </body>
</HTML>
```

Fig. 20.9 ASPX listing for the guestbook page (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

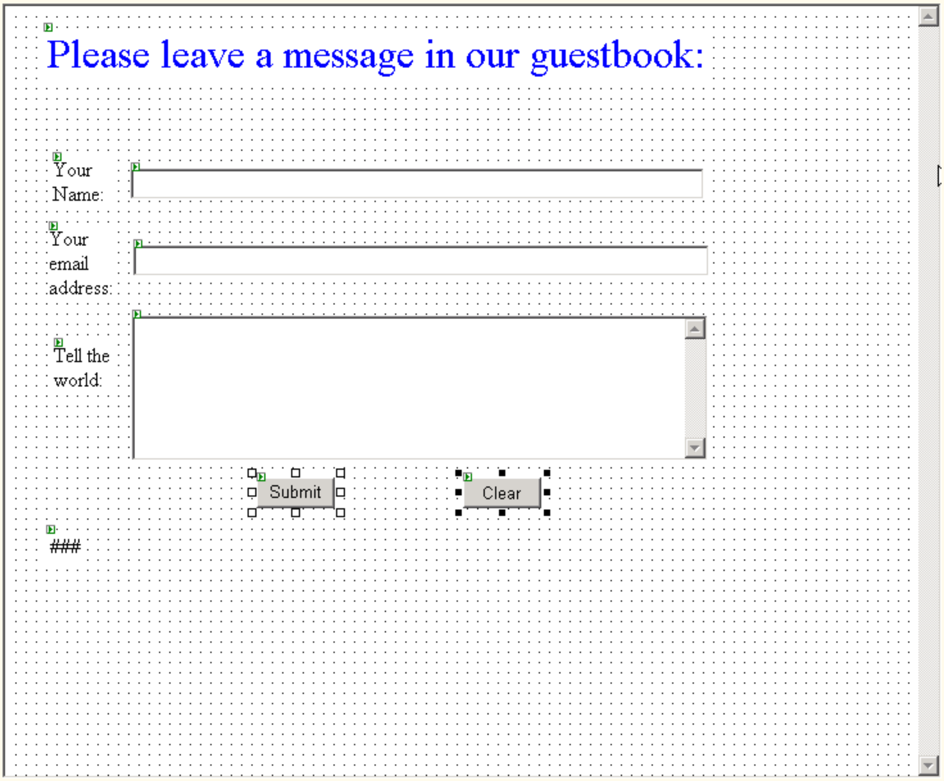


Fig. 20.9 ASPX listing for the guestbook page (part 3 of 3).

The screen capture in Fig. 20.9 shows what generated the ASP.NET code seen. The top item is in fact a **`Table`** called **`greetingTable`**. This will display a thank you message and a horizontal rule after a message has been left. The rest are **`TextBox`**es and **`Label`**s. The two buttons require event handlers which can be created by double clicking on them. The event handlers will be in the code-behind file, the listing for which is in Fig. 20.10. Lastly, we have another **`Table`** called **`messageTable`** to display all guestbook entries.

```
// Fig. 20.10
// The code-behind file for the guestbook page.

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
```

Fig. 20.10 Code-behind file for the guestbook application (part 1 of 7).

```
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.IO;

namespace Guestbook
{
   public class GuestbookPage : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Table greetingTable;
      protected System.Web.UI.WebControls.Table messageTable;
      protected System.Web.UI.WebControls.Label Label1;
      protected System.Web.UI.WebControls.TextBox nameTextBox;
      protected System.Web.UI.WebControls.Label Label2;
      protected System.Web.UI.WebControls.TextBox emailTextBox;
      protected System.Web.UI.WebControls.TextBox
         messageTextBox;
      protected System.Web.UI.WebControls.Button Button1;
      protected System.Web.UI.WebControls.Button Button2;
      protected System.Web.UI.WebControls.Label Label3;

      public GuestbookPage()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      private void Page_Load( object sender,
         System.EventArgs e )
      {
      }

      private void Page_Init( object sender, EventArgs e )
      {
         InitializeComponent();
      }

      // Visual Studio generated code

      public void clearButton_Click( object sender,
         System.EventArgs e )
      {
         nameTextBox.Text = "";
         emailTextBox.Text = "";
         messageTextBox.Text = "";
      }

      public TableRow MakeHorizontalRule()
      {
         TableRow hRuleRow;
         TableCell hCell;
         HtmlGenericControl hRule;

         // Create Horizontal Rule
         hRule = new HtmlGenericControl();
         hRule.TagName = "HR";
```

Fig. 20.10 Code-behind file for the guestbook application (part 2 of 7).

```

         // Create a row for the Horizontal Rule
         hRuleRow = new TableRow();
         hCell = new TableCell();
         hCell.Controls.Add( hRule );
         hRuleRow.Cells.Add( hCell );

         return hRuleRow;
      }

      public void FillMessageTable()
      {
         TableRow row;
         TableCell cell;
         HyperLink h;

         // Open guestbook file for reading
         StreamReader reader = new StreamReader(
            @"C:\Inetpub\wwwroot\GuestBook\guestbook.txt");

         // Create and add Header row
         row = new TableRow();
         cell = new TableCell();
         cell.Controls.Add(
            new LiteralControl( "Guestbook Entries:" ) );
         row.Cells.Add( cell );
         messageTable.Rows.Add( row );

         // Add Horizontal Rule
         messageTable.Rows.Add( MakeHorizontalRule() );

         try
         {
            while ( true )
            {
               // Read in one line from file
               string message = reader.ReadLine();

               // Throw exception if reached end of file
               if ( message == null )
                  throw new IOException();

               // Split the string into its four parts
               char[] separator = { '\t' };
               string[] parts = message.Split( separator );

               row = new TableRow();
               cell = new TableCell();

               // Create Hyperlink for email
               h = new HyperLink();
               h.NavigateUrl = "mailto:" + parts[ 2 ];
               h.Text = parts[ 1 ];

```

Fig. 20.10 Code-behind file for the guestbook application (part 3 of 7).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

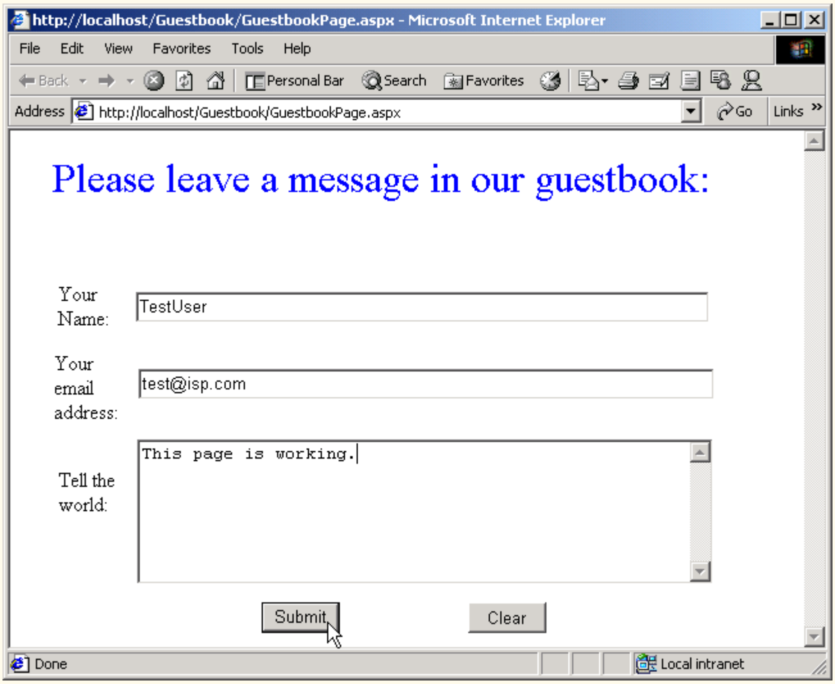
```
                  // Add the date, hyperlink, and message to the
                  // cell
                  cell.Controls.Add(
                     new LiteralControl( parts[ 0 ] + " " ) );
                  cell.Controls.Add( h );
                  cell.Controls.Add(
                     new LiteralControl( ": " + parts[ 3 ] ) );

                  // Add cell to row, and row to the table
                  row.Cells.Add( cell );
                  messageTable.Rows.Add( row );

                  // Add Horizontal Rule
                  messageTable.Rows.Add( MakeHorizontalRule() );
               }
            }
            catch( IOException e )
            {
            }

            reader.Close();
         }

         public void FillGreetingTable()
         {
            // Get the Hyperlink from the last message posted
            HyperLink h = (HyperLink)messageTable.Rows[
               messageTable.Rows.Count - 2 ].Cells[
               0 ].Controls[ 1 ];
            TableCell cell = new TableCell();
            TableRow row = new TableRow();

            // Create thank you message using the text from the
            // hyperlink from the last message posted
            cell.Controls.Add(
               new LiteralControl( "Thanks for your entry, " +
               h.Text + "!" ) );
            cell.Font.Size = 24;
            cell.ForeColor = Color.Blue;

            // Add the cell with the message to the row, and
            // the row to the greetingTable.  Then add Horizontal
            // Rule.
            row.Cells.Add( cell );
            greetingTable.Rows.AddAt( 0, row );
            greetingTable.Rows.AddAt( 1, MakeHorizontalRule() );
         }

         public void submitButton_Click( object sender,
            System.EventArgs e )
         {
            // Open or create file
            FileStream guestbookFile = new FileStream(
               @"C:\Inetpub\wwwroot\GuestBook\guestbook.txt",
```

Fig. 20.10 Code-behind file for the guestbook application (part 4 of 7).

```
               FileMode.OpenOrCreate );

            // Open streams for writing and reading file
            StreamWriter guestbook =
               new StreamWriter( guestbookFile );
            StreamReader getToEnd=
               new StreamReader( guestbookFile );

            // Get to the end of the file
            getToEnd.ReadToEnd();

            // Write new message to file
            guestbook.WriteLine(
               DateTime.Now.Date.ToString().Substring( 0, 10 ) +
               "\t" + nameTextBox.Text + "\t" + emailTextBox.Text
               + "\t" + messageTextBox.Text );

            // Clear all textboxes and close all streams
            nameTextBox.Text = "";
            emailTextBox.Text = "";
            messageTextBox.Text = "";
            guestbook.Close();
            getToEnd.Close();
            guestbookFile.Close();

            // Fill the Tables
            FillMessageTable();
            FillGreetingTable();
         } // end of submitButton_Click
      }
   }
```

Fig. 20.10 Code-behind file for the guestbook application (part 5 of 7).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01


http://localhost/Guestbook/GuestbookPage.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Favorites
Address
http://localhost/Guestbook/GuestbookPage.aspx
Go
Links
Submit
Clear
Done
Local intranet

Please leave a message in our guestbook:

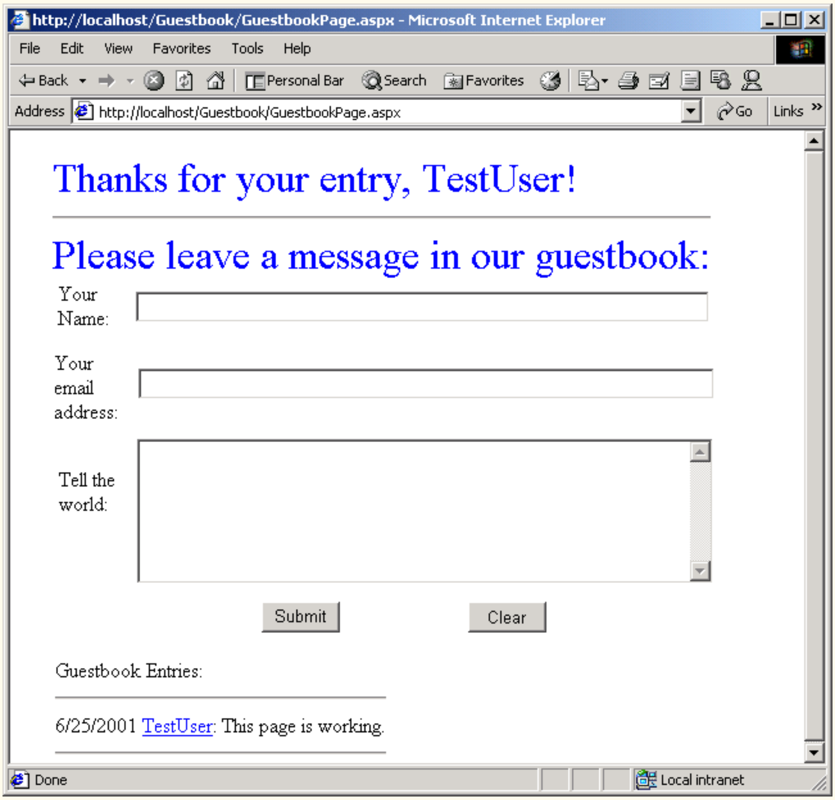
Your Name: TestUser

Your email address: test@isp.com

Tell the world: This page is working.

Fig. 20.10 Code-behind file for the guestbook application (part 6 of 7).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01


http://localhost/Guestbook/GuestbookPage.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/Guestbook/GuestbookPage.aspx
Go Links
Submit
Clear
Done
Local intranet

Thanks for your entry, TestUser!

Please leave a message in our guestbook:

Your Name:

Your email address:

Tell the world:

Guestbook Entries:

6/25/2001 TestUser: This page is working.

Fig. 20.10 Code-behind file for the guestbook application (part 7 of 7).

Lines 49–55 constitute the event handler for **`clearButton`** and simply sets all the **`TextBox`**es to contain no text. Lines 166–200 are the code for the event handler for **`submitButton`**. Lines 172–174 create a **`FileStream`** referencing the file containing all the guestbook entries. Then lines 177–180 use this **`FileStream`** to create a **`StreamWriter`** and **`StreamReader`** to read and write text from and to the file. Line 190 uses **`StreamReader`**’s method **`ReadToEnd`** to read the entire file. This places the pointer in the file after the last character in the file. So when lines 186–189 execute, the file is appended with the appropriate message. Before the event handler exits, it calls two methods: **`FillMessageTable`** and **`FillGreetingTable`**.

`FillMessageTable` as its name suggest places all the guestbook entries in **`messageTable`**. Any object of type **`Table`** consists of a set of rows of type **`TableRow`** which in turn consists of a set of cells of type **`TableCell`**. Any object that we want to place in a **`Table`** must first be placed in a **`TableCell`**, which needs to be placed in a **`TableRow`**, which in turn needs to be placed in the **`Table`**.

`FillMessageTable` starts out creating a row containing only the words **`"Guestbook Entries"`**. Notice the use of an unfamiliar class ***`LiteralControl`***—a control that is usually shown as plain text on a page. We then add a row containing the return value of method **`MakeHorizontalRule`**. **`MakeHorizontalRule`** is a method that returns

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

a single **TableRow** containing one **TableCell** which has in it an object of type ***HtmlGenericControl***. **HtmlGenericControl** represents any HTML control. We make it into a horizontal rule on line 65 when we assign its **TagName** property the value **"HR"**, i.e. a horizontal rule. Lines 68–71 place this **HtmlGenericControl** into a **TableCell** which is placed into a **TableRow**. We are going to separate all the guestbook entries with a horizontal rule so this method will be useful. **FillMessageTable** then begins executing an infinite loop which reads in a line from the guestbook file, if nothing has been read, line 106 throws a new **IOException** which is caught in the **catch** clause. Line 110 breaks up the read **string** into its individual pieces which are separated by tabs. Lines 112–113 create a new **TableRow** and **TableCell** object for this message. We can now begin constructing the row for this message.

Line 116 creates an object of type **HyperLink** to allow users to click on it and send e-mail to the person who left that message. Line 117 sets **h**'s **NavigateUrl** property to be an e-mail link to the e-mail provided by the user. Line 118 sets **h**'s **Text** property to be the name of the user. Anyone viewing the page will see the name of person who left the message on the page, but when that link is clicked a new e-mail will be created with the default e-mail client addressed to the e-mail address the poster provided. Lines 122–123 add a new **LiteralControl** to the **TableCell** containing the date the poster left the message–the first piece of information. Line 124 adds **HyperLink h** to the cell, and lines 125–126 add another **LiteralControl** containing the message itself to the **TableCell**. We now add this **TableCell** to our **TableRow** on line 136 and the **TableRow** to the **messageTable** on line 126. Line 133 makes use of method **MakeHorizontalRule** once more to add a row containing a horizontal rule to **messageTable**.

Method **FillGreetingTable** is invoked on line 201. Lines 146-148 of method **FillGreetingTable** get the **HyperLink** object of the last message in **messageTable**, i.e. the message of the person who just left it. Lines 154-156 create a **TableCell** containing a thank you message to the person who just left the message. This **TableCell** is then placed in a **TableRow** which is added to **greetingTable** using **Table** method **AddAt**. **AddAt** takes two arguments—the first is an integer specifying the index at which to add the row, and the second which is the **TableRow** to add.

20.6.2 Case Study: An Instant Page Content Builder

This case study presents a complex Web Application consisting of four linked ASP.NET pages. This application starts on **login.aspx** where users select their name from the drop–down list and enter their password. The information provided is sent on to the next page—**submitlogin.aspx**. If the password provided is the same as the password in the database the user is allowed to proceed to **instantpage.aspx**; otherwise the user is sent back to **login.aspx** and instructed to enter the correct password. **instantpage.aspx** allows users to enter their name, the name of the file they wish to create, the title of their document and its contents. The information entered here is sent on to **process.aspx** which processes this information and creates an ASP.NET page with the corresponding properties. What the users see once they get to **process.aspx** is a page with a link to their newly created file and a time. Clicking on the link takes them to their new page which includes a time stamp of the last request for the page.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

The ASPX listing for the first page—**`login.aspx`**—is shown in Fig. 20.11.

```
<%-- Fig. 20.10 --%>
<%-- ASPX code for login form. --%>

<%@ Page language="c#" Codebehind="login.cs"
    AutoEventWireup="false"
    Inherits="PageBuilder.LoginPage" %>
<html>
<head>
   <meta name=vs_targetSchema content="HTML 4.0">
   <meta name=vs_showGrid content="True">
   <meta name="GENERATOR"
         Content="Microsoft Visual Studio 7.0">
   <meta name="CODE_LANGUAGE" Content="C#">
</head>
  <body ms_positioning="GridLayout">

<!-- #include virtual = "b2bheader.shtml" -->
<br>
<%
   if ( Session[ "loginFailure" ] == null )
      Session.Add( "loginFailure", false );

   if ( (bool)Session[ "loginFailure" ] == true )
      Response.Write( "<FONT SIZE = \"4\" COLOR = \"red\">" +
         "Login attempt failed, please try again<P></FONT>" );
%>
<!-- Visual Studio Generated code for the WebForm -->
<form action="submitlogin.aspx" method="post" runat="server">
   <asp:DropDownList id=nameList runat="server" height="22"
        width="164" style="Z-INDEX: 101; LEFT: 101px;
        POSITION: absolute; TOP: 207px">
   </asp:DropDownList>
   <asp:Label id=Label2 runat="server" style="Z-INDEX: 102;
        LEFT: 13px; POSITION: absolute; TOP: 207px">Name:
   </asp:Label>
   <asp:Label id=Label3 runat="server" style="Z-INDEX: 104;
        LEFT: 13px; POSITION: absolute; TOP: 271px">Password:
   </asp:Label>
   <asp:Button id=Button1 runat="server" Text="Log Me In"
        Height="24" Width="108" style="Z-INDEX: 105;
        LEFT: 101px; POSITION: absolute; TOP: 319px">
   </asp:Button>
   <asp:Label id=Label1 runat="server" Font-Size="Small"
        style="Z-INDEX: 106; LEFT: 13px; POSITION: absolute;
        TOP: 159px">Please select your name and enter your
          password to login:
   </asp:Label>
   <asp:TextBox id=password style="Z-INDEX: 107; LEFT: 104px;
        POSITION: absolute; TOP: 265px" runat="server"
        Width="161" Height="24" TextMode="Password">
   </asp:TextBox>
```

Fig. 20.11 ASPX listing for log in page (part 1 of 2)

```
52  </form>
53
54  <!-- #include virtual = "b2bfooter.shtml" -->
55
56  </body></html>
```

Fig. 20.11 ASPX listing for log in page (part 2 of 2)

Line 17 is a *server side include* (*SSI*) statement that incorporates the contents of **`b2bheader.shtml`** (Figure 20.20) into the ASPX file. Server-side includes are commands embedded in HTML documents that add dynamic content. The SSI statement in line 14 is replaced with the contents of the file **`b2bheader.shtml`**. Not all Web servers support the available SSI commands. Therefore, SSI commands are written as HTML comments. SSI statements always execute before any scripting code executes.

We also use an SSI in line 54 to include **`b2bfooter.shtml`** (Figure 20.21). The word ***`virtual`*** in the SSI refers to the include file's path as it appears below the server's root directory. This is often referred to as a *virtual path*. SSIs can also use ***`file`*** instead of **`virtual`** to indicate a *physical path* on the server.

The expression on line 17 uses **`HttpSessionState`**'s indexer to get the value of the entry called "**`loginFailure`**". There is another version of the indexer which takes an integer parameter instead of a string. If the variable named "**`loginFailure`**" exists the indexer will return its values; otherwise it will return **`null`**. This line checks to see if it exists, if not it will add a session variable called "**`loginFailure`**" having the value **`false`** on line 18 using **`HttpSessionState`** method **`Add`**. The first parameter to **`Add`** is a **`string`** denoting the name of the variable, and the second is any **`object`** which is the value of the variable. Line 20 also looks up the value of session variable "**`loginFailure`**". We know that this variable is of type **`bool`**, and if it did not exist before, it was just created in preceding if statement, so we cast to a **`bool`**. If it is true then the user did not provide a correct password and we must use the **`Response`** variable's **`Write`** method to send HTML code to the clients browser. In this case we send red text indicating there was a log in problem.

The ASP.NET elements define a **`DropDownList`** named **`nameList`** which will be populated in **`login.aspx.cs`** with user names from a database, a **`TextBox`** in which the users will enter their password, and some **`Label`**s. The code-behind file for this page is show in Fig. 20.12.

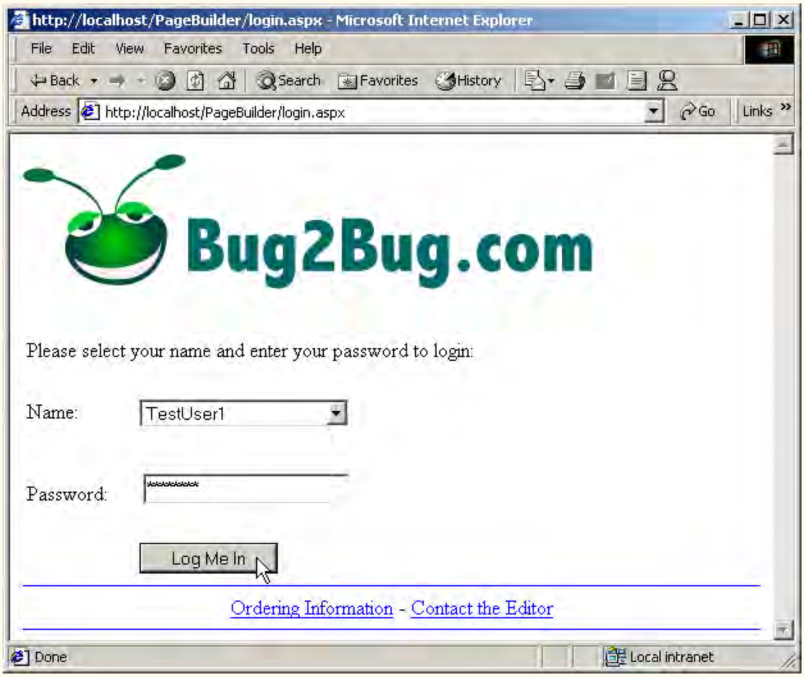
```
// Fig. 20.11
// code for login.cs

namespace PageBuilder
{
   using System;
   using System.Collections;
   using System.ComponentModel;
   using System.Data;
   using System.Drawing;
   using System.Web;
   using System.Web.SessionState;
```

Fig. 20.12 Code listing for login.cs (part 1 of 3).

```
   using System.Web.UI;
   using System.Web.UI.WebControls;
   using System.Web.UI.HtmlControls;
   using System.Data.ADO;

   public class LoginPage : System.Web.UI.Page
   {
      protected System.Data.ADO.ADODataSetCommand
         adoDataSetCommand1;
      protected System.Data.ADO.ADOConnection adoConnection1;
      protected System.Web.UI.HtmlControls.HtmlGenericControl
         H21;
      protected System.Web.UI.WebControls.TextBox password;
      protected System.Web.UI.WebControls.Label Label1;
      protected System.Web.UI.WebControls.Button Button1;
      protected System.Web.UI.WebControls.Label Label3;
      protected System.Web.UI.WebControls.Label Label2;
      protected System.Web.UI.WebControls.DropDownList
         nameList;
      protected ADODataReader dataReader;

      public LoginPage()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      protected void Page_Load( object sender, EventArgs e )
      {
      }

      protected void Page_Init( object sender, EventArgs e )
      {
         InitializeComponent();

         // open database connection
         adoConnection1.Open();

         // execute query
         adoDataSetCommand1.SelectCommand.Execute(
            out dataReader );

         // while we can read a row from the result of the
         // query, add the first item to the dropdown list
         while ( dataReader.Read() )
            nameList.Items.Add( dataReader.GetString( 0 ) );
      }

      private void InitializeComponent()
      {
        // Visual Studio generated code
      }
   }
}
```

Fig. 20.12 Code listing for login.cs (part 2 of 3).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01


http://localhost/PageBuilder/login.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
http://localhost/PageBuilder/login.aspx
Go
Links
Bug2Bug.com
Log Me In
Done
Local intranet

Please select your name and enter your password to login:

Name: TestUser1

Password: ********

Ordering Information - Contact the Editor

Fig. 20.12 Code listing for login.cs (part 3 of 3).

We want to use an *Access* database to get our user names and populate **`nameList`**. We follow the same steps to add **`login.mdb`** as a data connection in **Server Explorer** and configure the **`ADODataSetCommand`** as in Chapter 19. We want the select clause to be "**`SELECT loginID FROM Users`**" to indicate that we want all the names stored in table Users. This constitutes most of the code in **`InitializeComponent`** which we omit for presentation purposes.

The **`while`** loop lines 56–57 places every user name in the database into **`nameList`**.

When the user clicks **Log Me In**, the browser is redirected to **`submitlogin.aspx`**—the code which is shown in Fig. 20.13.

```
<%-- Fig. 20.12 --%>
<%-- ASPX code for the page that process the login information. --%>

<%@ Page language="c#" Codebehind="SubmitLogin.cs"
    AutoEventWireup="false"
    Inherits="PageBuilder.SubmitLogin" %>

<html>
  <head>
    <meta name="GENERATOR"
```

Fig. 20.13 ASPX listing for submitlogin.aspx (part 1 of 2)

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
            Content="Microsoft Visual Studio 7.0">
        <meta name="CODE_LANGUAGE" Content="C#">
    </head>
    <body>
        <form method="post" runat="server" ID=Form1>
        </form>
</body>
</html>
```

Fig. 20.13 ASPX listing for submitlogin.aspx (part 2 of 2)

As you see **submitlogin.aspx** does not display anything, the code-behind file **submitlogin.cs** (Fig. 20.14) performs the work involved in verifying the user's password.

```
// Fig. 20.13
// code for submitlogin.cs

namespace PageBuilder
{
   using System;
   using System.Collections;
   using System.ComponentModel;
   using System.Data;
   using System.Drawing;
   using System.Web;
   using System.Web.SessionState;
   using System.Web.UI;
   using System.Web.UI.WebControls;
   using System.Web.UI.HtmlControls;
   using System.Data.ADO;

   public class SubmitLogin : System.Web.UI.Page
   {
      protected System.Data.ADO.ADOConnection
         adoConnection1;
      protected System.Data.ADO.ADODataSetCommand
         adoDataSetCommand1;

      public SubmitLogin()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      protected void Page_Load( object sender, EventArgs e )
      {
         ADODataReader dataReader;

         // create a connection to the database
         adoConnection1.Open();

         // create a DataSetCommand and set the select command
```

Fig. 20.14 Code-behind file for submitlogin.cs (part 1 of 2)

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
38          // to find the password of the username from the
39          // dropdown list
40          adoDataSetCommand1.SelectCommand = new ADOCommand(
41             "SELECT * FROM Users WHERE loginID = '" +
42             Request.Form.Get( "nameList" ).ToString() +
43             "'", adoConnection1 );
44
45          adoDataSetCommand1.SelectCommand.Execute(
46             out dataReader );
47
48          dataReader.Read();
49
50          // if the password the user provided the correct
51          // password direct them to instantpage.aspx,
52          // otherwise send them back to login.aspx with
53          // "loginFailure set to true
54          if ( Request.Form.Get( "password" ).ToString() ==
55             dataReader.GetString( 1 ) )
56          {
57             Session[ "loginFailure" ] = false;
58             Response.Redirect(
59                "instantpage.aspx" );
60          }
61          else
62          {
63             Session[ "loginFailure" ] = true;
64             Response.Redirect( "login.aspx" );
65          }
66       }
67
68       protected void Page_Init( object sender, EventArgs e )
69       {
70          InitializeComponent();
71       }
72
73       private void InitializeComponent()
74       {
75          // Visual Studio generated code
76       }
77    }
78 }
```

Fig. 20.14 Code-behind file for submitlogin.cs (part 2 of 2)

For this page as well, we drag the **Users** table from our database in **Server Explorer**. We place all our password checking code in the **Page_Load** event handler. Lines 40–43 set the **CommandText** property of the **SelectCommand** property of **adoDataSetCommand1** to an SQL query that retrieves the password of the user with **loginID** that was selected from **nameList**. The **if** statement lines 54–66 sends the user back **login.aspx** if the password did not match what is in the database, and sends them to **instantpage.aspx** if it did.

Once users successfully select their user name and enter the correct password they are send to **instantpage.aspx** to start building their custom page. The ASPX listing for **instantpage.aspx** is shown in Fig. 20.15.

```
<%-- Fig. 20.14 --%>
<%-- Web Form for user input regarding the page they --%>
<%-- wish to create. --%>

<%@ Page language="c#" Codebehind="instantpage.cs"
    AutoEventWireup="false" Inherits="PageBuilder.InstantPage" %>
<html><head>
<TITLE>Instant Page Content Builder</TITLE>

<meta content="HTML 4.0" name=vs_targetSchema>
<meta content=True name=vs_showGrid>
<meta content="Microsoft Visual Studio 7.0" name=GENERATOR>
<meta content=C# name=CODE_LANGUAGE></head>
<body ms_positioning="GridLayout">

<!-- #include virtual = "b2bheader.shtml" -->

<h2 id=H21 runat="server"> &nbsp; </h2><br>

<%
   if ( Session[ "errorMessage" ] == null )
      Session.Add( "errorMessage", "" );

   if ( Session[ "welcomeBack" ] == null )
      Session.Add( "welcomeBack", "" );

   if ( Request.QueryString.Get("error") == "yes" )
      Response.Write( Session[ "errorMessage" ].ToString() );
   else
      Response.Write( Session[ "welcomeBack"].ToString() );
%>

<form  action="process.aspx" method=post runat="server">
<asp:TextBox id=name style="Z-INDEX: 101; LEFT: 117px;
     POSITION: absolute; TOP: 238px" runat="server"
     width="464" height="24">
</asp:TextBox>
<asp:TextBox id=filename style="Z-INDEX: 102; LEFT: 117px;
     POSITION: absolute; TOP: 286px" runat="server"
     width="464" height="24">YourFileName.aspx
</asp:TextBox>
<asp:TextBox id=doctitle style="Z-INDEX: 103; LEFT: 117px;
     POSITION: absolute; TOP: 334px" runat="server"
     width="464" height="24">Document Title
</asp:TextBox>
<asp:TextBox id=content style="Z-INDEX: 109; LEFT: 117px;
     POSITION: absolute; TOP: 382px" runat="server"
     width="466" height="136" textmode="MultiLine">
Replace this text with the information you would like to post.
```

Fig. 20.15 ASPX listing for instantpage.aspx (part 1 of 2).

```
50  </asp:TextBox>
51  <asp:Button  id=submitButton style="Z-INDEX: 104;
52       LEFT: 181px;
53       POSITION: absolute;
54       TOP: 534px"
55       runat="server" Text="Submit">
56  </asp:Button>
57  </form>
58
59  <asp:Label id=Label1 style="Z-INDEX: 105; LEFT: 29px;
60       POSITION: absolute; TOP: 230px" runat="server"
61       Width="58" Height="16">Your Name:
62  </asp:Label>
63  <asp:Label id=Label2 style="Z-INDEX: 106; LEFT: 29px;
64       POSITION: absolute; TOP: 278px" runat="server"
65       Width="66" Height="52">Enter the Filename:
66  </asp:Label>
67  <asp:Label id=Label3 style="Z-INDEX: 107; LEFT: 29px;
68       POSITION: absolute; TOP: 326px" runat="server"
69       Width="57" Height="40">Enter the Title:
70  </asp:Label>
71  <asp:Label id=Label4 style="Z-INDEX: 108; LEFT: 29px;
72       POSITION: absolute; TOP: 398px" runat="server"
73       Width="64" Height="40">Enter the Content:
74  </asp:Label>
75
76  <form method=post >
77  <asp:Button id=resetButton style="Z-INDEX: 110; LEFT: 381px;
78       POSITION: absolute; TOP: 534px" runat="server"
79       Text="Reset" >
80  </asp:Button>
81  </form>
82
83  <!-- #include virtual = "b2bfooter.shtml" -->
84  <div style="Z-INDEX: 111; LEFT: 31px; POSITION: absolute;
85       TOP: 130px"
86  ms_positioning="text2D"><font size=5>
87  Instant Page Content Builder</font>
88  </div>
89
90  </body></html>
```

Fig. 20.15 ASPX listing for instantpage.aspx (part 2 of 2).

Lines 19–30 constitute a client-side script which adds the **`errorMessage`** and **`welcomeBack`** session variables if they did not already exist. These variables will be used to hold the error and welcome back messages respectively, and will be set if necessary later on. Line 26 checks if the **`QueryString`** property of the **`Request`** object has a variable named **`error`**. A query string appears at the end of the HTTP address in your browser's address bar and has the form **`"?var-name=var-value"`**. If **`error`** has the value **`yes`** then we need to print an error message, otherwise we can print the **`welcomeBack`** message. If this is the first time we are at this page then the **`welcomeBack`** session variable has just been created and holds an empty **`string`**. The rest of the code creates the various

ASP.NET form elements required for the page—four **TextBox**es, four **Label**s and two **Button**s. In this page as well we include **b2bheader.shtml** and **b2bfooter.shtml** on lines 15 and 82 respectively.

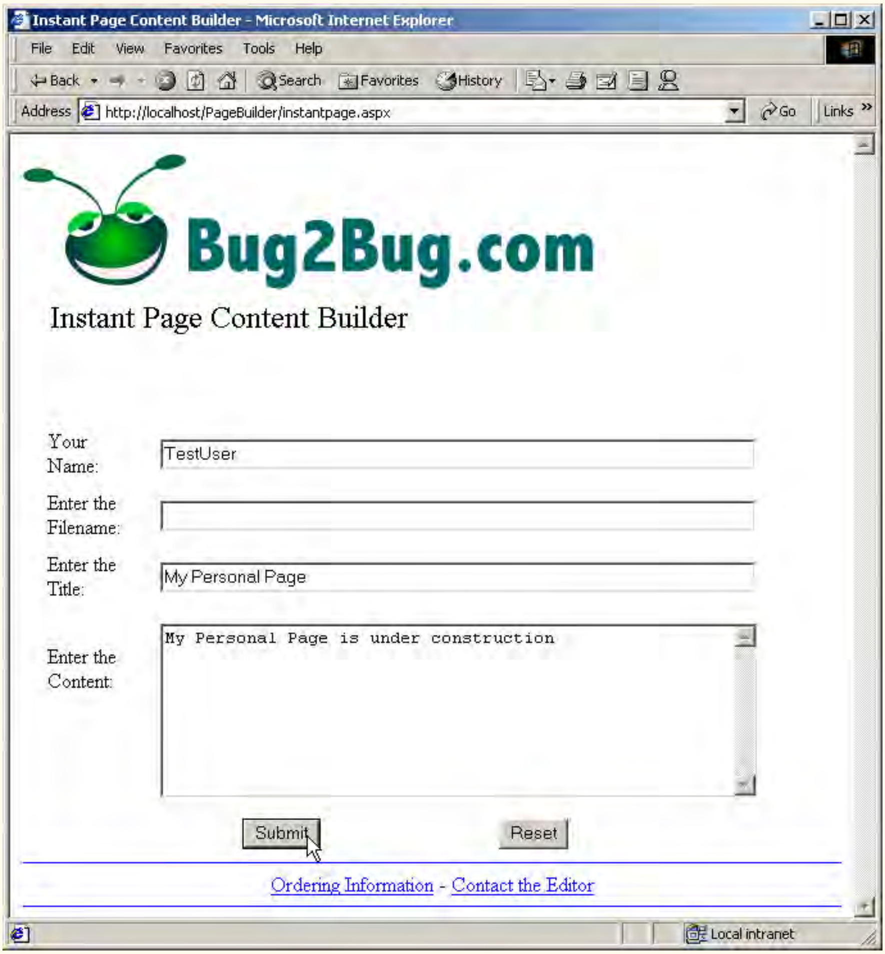
When the user clicks on either button the browser is redirected to **process.aspx**. This is specified in the action attribute of the form on line 32. **process.aspx** makes sure it has all the necessary information and redirects the user back to **instantpage.aspx** if necessary. The code listing for **instantpage.cs** is in Fig. 20.16 and has been formatted and cleaned up for presentation purposes, but not changed otherwise.

```
// Fig. 20.15
// code for instantpage.cs

namespace PageBuilder
{
   using System;
   using System.Collections;
   using System.ComponentModel;
   using System.Data;
   using System.Drawing;
   using System.Web;
   using System.Web.SessionState;
   using System.Web.UI;
   using System.Web.UI.WebControls;
   using System.Web.UI.HtmlControls;

   public class InstantPage : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.Button resetButton;
      protected System.Web.UI.WebControls.Label Label4;
      protected System.Web.UI.WebControls.Label Label3;
      protected System.Web.UI.WebControls.Label Label2;
      protected System.Web.UI.WebControls.Label Label1;
      protected System.Web.UI.WebControls.Button submitButton;
      protected System.Web.UI.WebControls.TextBox content;
      protected System.Web.UI.WebControls.TextBox doctitle;
      protected System.Web.UI.WebControls.TextBox filename;
      protected System.Web.UI.WebControls.TextBox name;

      public InstantPage()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }

      protected void Page_Load( object sender, EventArgs e )
      {
      }

      protected void Page_Init( object sender, EventArgs e )
      {
         InitializeComponent();
      }

```

Fig. 20.16 Code-behind file instantpage.cs (part 1 of 4)

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
      private void InitializeComponent()
      {
         this.Load +=
            new System.EventHandler ( this.Page_Load );
      }
   }
}
```


Instant Page Content Builder - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/PageBuilder/instantpage.aspx
Go Links
Bug2Bug.com
Local intranet

Instant Page Content Builder

Your Name: TestUser

Enter the Filename:

Enter the Title: My Personal Page

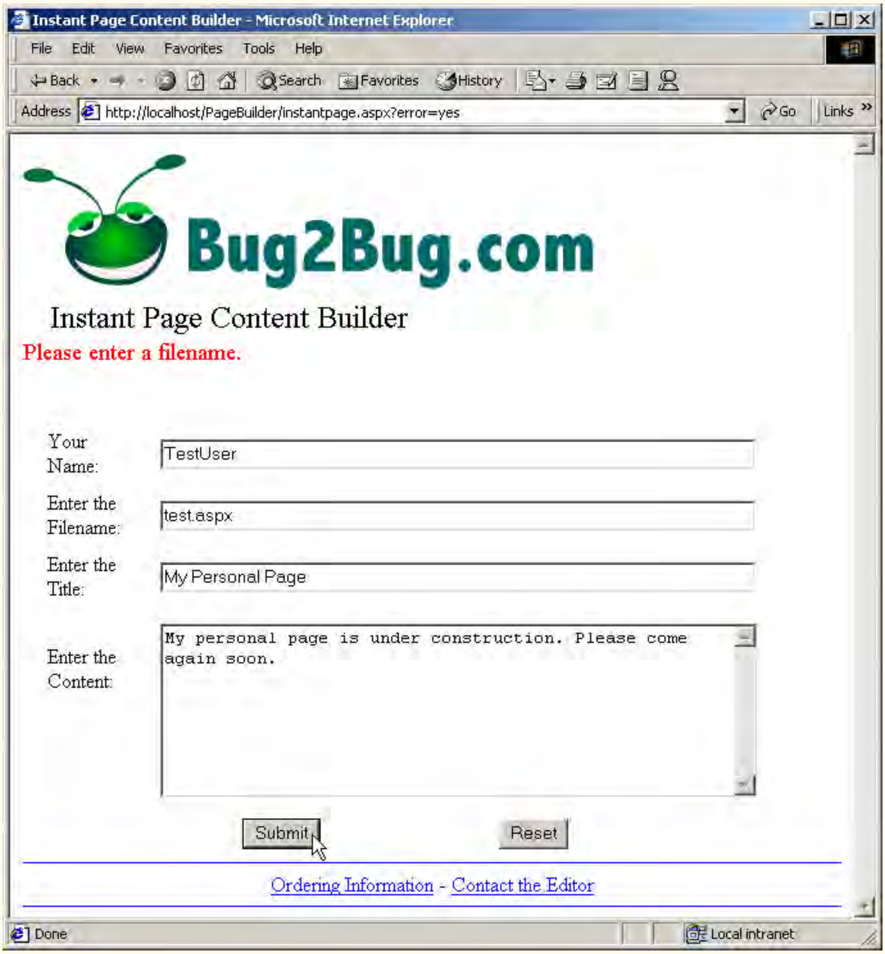
Enter the Content: My Personal Page is under construction

Submit Reset

Ordering Information - Contact the Editor

Fig. 20.16 Code-behind file instantpage.cs (part 2 of 4)

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01


Instant Page Content Builder - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/PageBuilder/instantpage.aspx?error=yes
Go Links
Bug2Bug.com

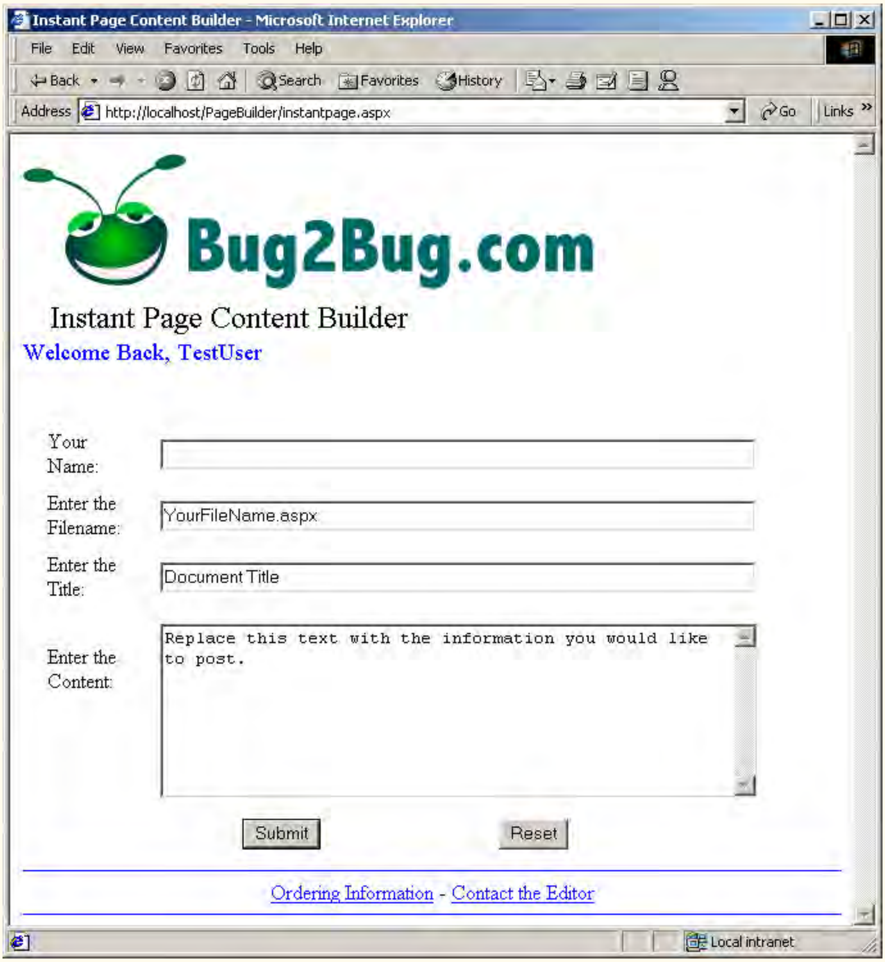
Instant Page Content Builder

Please enter a filename.

Your Name:	TestUser
Enter the Filename:	test.aspx
Enter the Title:	My Personal Page
Enter the Content:	My personal page is under construction. Please come again soon.

Submit
Reset
Ordering Information - Contact the Editor
Done
Local intranet

Fig. 20.16 Code-behind file instantpage.cs (part 3 of 4)


Instant Page Content Builder - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/PageBuilder/instantpage.aspx
Go Links
Bug2Bug.com
Local intranet

Instant Page Content Builder

Welcome Back, TestUser

Your Name:

Enter the Filename: YourFileName.aspx

Enter the Title: Document Title

Enter the Content: Replace this text with the information you would like to post.

Submit
Reset
Ordering Information - Contact the Editor

Fig. 20.16 Code-behind file instantpage.cs (part 4 of 4)

Information from **`instantpage.aspx`** is sent to **`process.aspx`** (Fig. 20.17) for verification and file creation.

```
<%-- Fig. 20.16 --%>
<%-- ASPX code for the page that creates the user’s page. --%>

<%@ Page language="c#" Codebehind="process.cs"
    AutoEventWireup="false" Inherits="PageBuilder.process" %>

<html><head>
<TITLE>File Generated:
<% Response.Write( Request.Form.Get( "filename" ) ); %>
</TITLE>
```

Fig. 20.17 ASPX listing for process.aspx (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
11     <meta name="GENERATOR"
12           Content="Microsoft Visual Studio 7.0">
13     <meta name="CODE_LANGUAGE" Content="C#"></head>
14 <BODY >
15 <!-- #include virtual = "b2bheader.shtml" -->
16 <center><h2><u>
17 <font face = "Arial" size = "3">
18    File generated:
19    <% Response.Write( Request.Form.Get( "filename" ) ); %>
20 </h2></u></center><br>
21 Your file is ready:
22 <a href = "userpages/
23 <% Response.Write( Request.Form.Get( "filename" ) ); %>">
24 <% Response.Write( Request.Form.Get( "doctitle" ) ); %></a>
25 
26 </font>
27     <form method="post" runat="server">&nbsp;
28     </form>
29 <!-- #include virtual = "b2bfooter.shtml" -->
30   </BODY></HTML>
```

Fig. 20.17 ASPX listing for process.aspx (part 2 of 2).

This page creates a page with a link to the user's file. Notice the use of client-side scripts throughout the page. Line 9 dynamically creates the title of the page as seen in the title bar of the client's browser. Lines 22–24 create a link to the user's new page with the name of the file as text for the link. The user's file itself is created in the code-behind file (Fig. 20.18).

```
// Fig. 20.17
// code for process.cs

namespace PageBuilder
{
   using System;
   using System.Collections;
   using System.ComponentModel;
   using System.Data;
   using System.Drawing;
   using System.Web;
   using System.Web.SessionState;
   using System.Web.UI;
   using System.Web.UI.WebControls;
   using System.Web.UI.HtmlControls;
   using System.IO;

   public class process : System.Web.UI.Page
   {
      public process()
      {
         Page.Init += new System.EventHandler( Page_Init );
      }
```

Fig. 20.18 Code-behind file for process.cs (part 1 of 4).

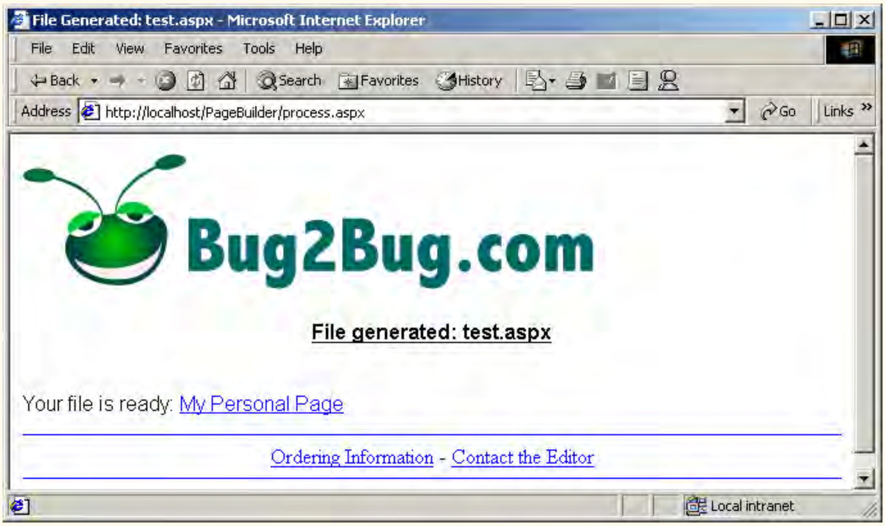
```

      protected void Page_Load( object sender, EventArgs e )
      {
         // if the user kept the defaul file name or left the
         // field empty tell the user they need to enter a
         // filename and redirect them back to instantpage.aspx
         if ( Request.Form.Get( "filename" ) ==
            "YourFileName.aspx" ||
            Request.Form.Get( "filename" ) == "" )
         {
            Session[ "errorMessage" ] = "<FONT COLOR = " +
               "\"red\" SIZE = \"4\"> Please enter a filename."
               + "</FONT><BR>";
            Response.Redirect( "instantpage.aspx?error=yes" );
         }

         // if the file the user wanted to create already
         // exists signal the user that he/she needs to chose
         // another name and redirect them back to
         // instantpage.aspx
         if ( File.FileExists(
            @"C:\Inetpub\wwwroot\PageBuilder\userpages\" +
            Request.Form.Get( "filename" ) ) )
            {
               Session[ "errorMessage" ] = "<FONT COLOR = " +
                  "\"red\" SIZE = \"4\"> This file name is in"
                  + " use. Please enter another filename" +
                  "</FONT><BR>";
               Response.Redirect(
                  "instantpage.aspx?error=yes" );
            }

         // create a new file
         FileStream textFile = new
            FileStream( @"C:\Inetpub\wwwroot\PageBuilder\" +
            "userpages\\" +
            Request.Form.Get( "filename" ), FileMode.Create );

         // create a welcomeBack session variable containing
         // a message in blue welcoming back the user
         Session[ "welcomeBack" ] = "<FONT COLOR = " +
               "\"blue\" SIZE = \"4\">Welcome Back, " +
               Request.Form.Get( "name" ) + "</FONT><BR>";

         // define the opening and closing delimiters of
         // client-side script
         string openMark = "<" + "%", closeMark = "%" + ">";

         // create the header of the page
         string header = openMark + "@ Page language=\"c#\" "
            + closeMark + "\r\n" + openMark + " // " +
            Request.Form.Get( "filename" ) + " " + closeMark +
            "\r\n" +
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "
```

Fig. 20.18 Code-behind file for process.cs (part 2 of 4).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```
            + "Transitional//EN\">\r\n<HTML>\r\n<HEAD>\r\n" +
            "<META NAME = \"author\" CONTENT = \"" +
            Request.Form.Get( "name" ) + "\">\r\n<META NAME "
            + "= \"pubdate\" CONTENT = \"" +
            DateTime.Now.ToString().Substring( 0, 10 ) +
            "\">\r\n<TITLE>" + Request.Form.Get( "doctitle" )
            + "</TITLE>\r\n</HEAD>\r\n<BODY>\r\n<FONT FACE = "
            + "\"arial\" SIZE = \"3\" >\r\n<!-- #include file"
            + " = \"../b2bheader.shtml\"-->\r\n<CENTER><U>" +
            "<H2>" + Request.Form.Get( "doctitle" ) +
            "</H2></U>\r\n<BR>\r\n";

         // create the footer of the page
         string footer = "\r\n</CENTER><BR><BR><BR>\r\n";
         footer += "You have requested this page on ";
         footer += openMark;
         footer += " Response.Write( DateTime.Now.ToString()";
         footer += ".Substring( 0, 10 ) ); " + closeMark;
         footer += ",\r\nat " + openMark;
         footer += " Response.Write( DateTime.Now.ToString()";
         footer += ".Substring( 11, 8 ) );" + closeMark;
         footer += ".<BR>\r\n<!-- #include virtual = \"";
         footer += "../b2bfooter.shtml\" -->\r\n</FONT>\r\n";
         footer += "<BODY>\r\n</HTML>";

         // create a StreamWriter to the file
         StreamWriter writer = new StreamWriter( textFile );

         // write the header, the content the user wanted to
         // display, and the footer to the file
         writer.WriteLine( header );
         writer.WriteLine( Request.Form.Get( "content" ) );
         writer.Write( footer );

         writer.Close();
      }

      protected void Page_Init( object sender, EventArgs e )
      {
         InitializeComponent();
      }

      private void InitializeComponent()
      {
         this.Load +=
            new System.EventHandler ( this.Page_Load );
      }
   }
}
```

Fig. 20.18 Code-behind file for process.cs (part 3 of 4).


File Generated: test.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/PageBuilder/process.aspx
Go Links
Bug2Bug.com
Local intranet

File generated: test.aspx

Your file is ready: My Personal Page

Ordering Information - Contact the Editor

Fig. 20.18 Code-behind file for process.cs (part 4 of 4).

All the code for the generation of the user's file is placed in the **`Page_Load`** event handler. Lines 30–38 make sure that the user did not keep the default file name and did not leave the field empty. If this is the case however, session variable **`errorMessage`** is set appropriately and the user is redirected back to "**`instantpage.aspx?error=yes`**". **`errorMessage`** contains HTML code that when rendered in a browser will print out text of red color because we are using the **`HttpResponse Write`** method which outputs HTML code to the browser. Notice that there is a query string at the end of the page name to signal to **`instantpage.aspx`** that it needs to print the error message.

If the file the user wanted to create already exists the if statement on lines 44–54 sets **`errorMessage`** to the appropriate value and redirects the user back to **`instantpage.aspx`** with the query string "**`?error=yes`**".

If neither one of these was the case method continues to execute and creates the file with the user's chose name on lines 58–60. Lines 64–66 set the **`welcomeBack`** session variable to contain a welcome back message along with the user's name.

The **`string header`** defined in lines 73–88 contains the HTML code necessary for the part portion of the user's page. It creates all the opening HTML tags, the **`meta`** tags with the user's information, and places the name of the page in the middle. It also places an SSI include at the top of the file.

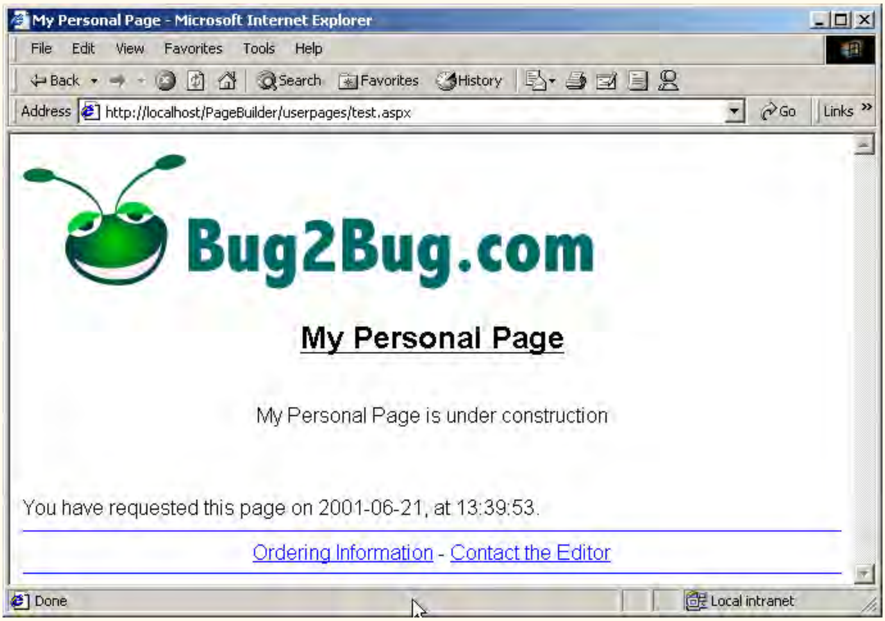
The **`string footer`** created in lines 91–101 creates the bottom portion of the page by including a script to print out the time the user accessed this page. **`b2bfooter.shtml`** is included as usual.

A **`StreamWriter`** to the newly created file is made so that the HTML code can be placed in the file. The **`header`**, the information the user wanted to display and the **`footer`** are written to the file. Now when the user clicks on the link generated by **`process.aspx`** the browser will be sent to the newly created page. Fig. 20.19 shows the code generated for a page and has a screen capture of what this page would look like as rendered by a browser.

```
<%-- Fig. 20.18 --%>
<%-- ASPX code for the generated page --%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<META NAME = "author" CONTENT = "TestUser">
<META NAME = "pubdate" CONTENT = "2001-05-30">
<TITLE>My Personal Page</TITLE>
</HEAD>
<BODY>
<FONT FACE = "arial" SIZE = "3" >

<img src="c:\inetpub\wwwroot\PageBuilder\bug2bug.gif">
<CENTER><U><H2>My Personal Page</H2></U>
<BR>

My personal page is under construction. Come again soon.

</CENTER><BR><BR><BR>
You have requested this page on 2001-05-30,
at 09:42:36.<BR>
<SPAN style="position:absolute; bottom:0px">
<HR COLOR = "blue" SIZE = "1">
<CENTER>
<A HREF = "mailto:orders">Ordering Information</A> -
<A HREF = "mailto:editor">Contact the Editor</A><BR>
<HR COLOR = "blue" SIZE = "1">
</CENTER>
</SPAN>
</FONT>
<BODY>
</HTML>
```

Fig. 20.19 A page created for a user (part 1 of 2).

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01


My Personal Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
Bug2Bug.com
Done
Local intranet

Address: http://localhost/PageBuilder/userpages/test.aspx

My Personal Page

My Personal Page is under construction

You have requested this page on 2001-06-21, at 13:39:53.

Ordering Information - Contact the Editor

Fig. 20.19 A page created for a user (part 2 of 2).

Notice that we use scripts to include the current date and time as well. This ASP.NET page has no code-behind file—this is because it does not need any additional code and we did not place an **`Inherits`** attribute at the top of the ASPX code. The HTML and scripts will be executed and rendered as usual.

```
<%-- Fig. 20.19 --%>
<%-- Listing for b2bheader.shtml --%>

<img src="c:\inetpub\wwwroot\PageBuilder\bug2bug.gif">
```

Fig. 20.20 HTML listing for b2bheader.shtml

Figure 20.20 is the code listing for **`b2bheader.shtml`**—it consists of only a tag to display an image. Figure 20.21 is the code listing for **`b2bfooter.shtml`**—two e-mail links with a horizontal rule above and below which are always placed at the bottom of the page. We do this because the ASP.NET elements have absolute positioning, and anything added dynamically or without absolute coordinates may overlap with them or cause the page to look strange. This ensures that no matter what the page is, the footer always appears at the bottom.

```
<%-- Fig. 20.20 --%>
<%-- Listing for b2bfooter.shtml --%>
```

Fig. 20.21 HTML listing for b2bfooter.shtml

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

```

<SPAN style="position:absolute; bottom:0px">
<HR COLOR = "blue" SIZE = "1">
<CENTER>
<A HREF = "mailto:orders">Ordering Information</A> -
<A HREF = "mailto:editor">Contact the Editor</A><BR>
<HR COLOR = "blue" SIZE = "1">
</CENTER>
</SPAN>
```

Fig. 20.21 HTML listing for b2bfooter.shtml

20.7 Other Topics in ASP.NET: Tracing and Cookieless Session State

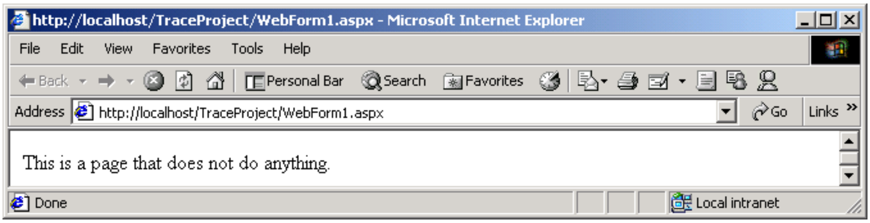
In this section we will discuss how to use ASP.NET features to help you design robust applications quicker and easier. We will start our discussion with Visual Studio.NET's tracing capabilities.

Debugging is one way to find the bugs in an application—*tracing* is another. Tracing is the placing of statements throughout a program that output information about its status during execution. The designer can then look at these statements to determine the problem. In Windows Applications one might use message boxes to trace a program; in ASP.NET one might use **`Response.Write`** to output information directly to the page to do the same. Using **`Response.Write`** for tracing in ASP.NET, however, has several drawbacks. One of these drawbacks is that once your application is working properly, you have to search for these statements in your program and remove them. This is time consuming and error-prone—you must differentiate between the statements that contribute to your code and those that you used for tracing.

Fortunately, ASP.NET has two more sophisticated forms of tracing: *page tracing* and *application tracing*. Page tracing involves tracing the actions of an individual page. Setting the ***`trace`*** property of the page to **`True`** in the **Properties** window of Visual Studio.NET turns on tracing for that page. Instead of using the **`Write`** method of the **`Response`** object, we will use the ***`Write`*** method of the ***`Trace`*** object. Object **`Trace`** is an instance of the ***`TraceContext`*** class and is used in ASP.NET to perform tracing. The **`Trace`** object also has method ***`Warn`*** which prints statements in red. When tracing is disabled on a page, the **`Trace`** object is disabled. This means that you do not have to go through your code and remove the write statements—simply set **`trace`** to **`False`**.

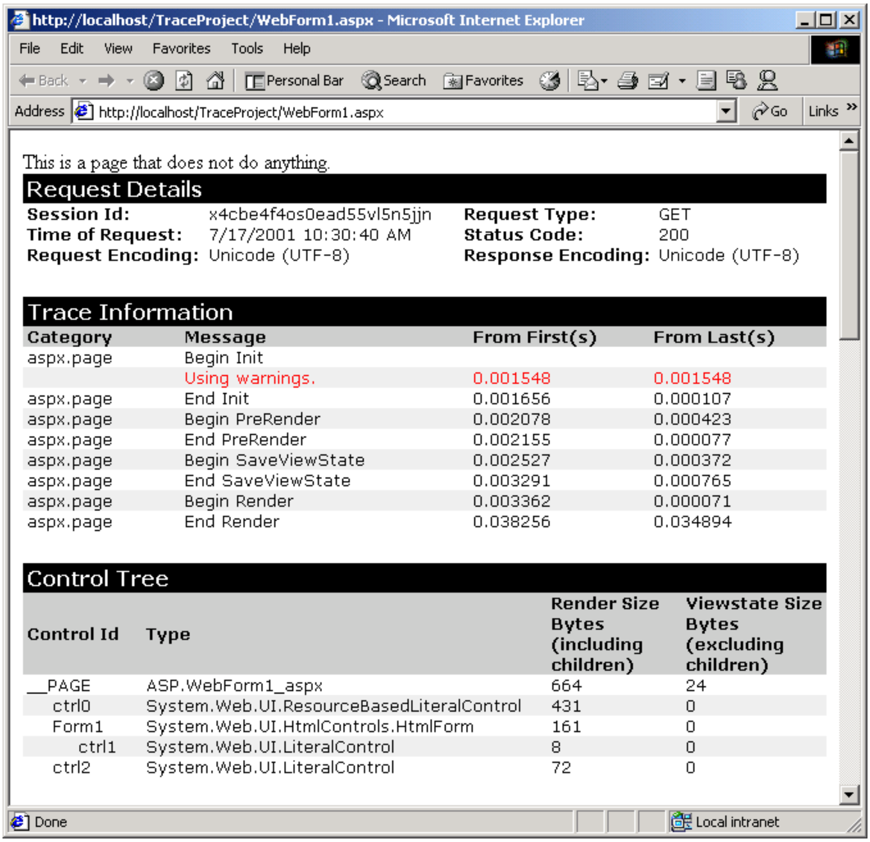
© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

Figure 20.22 shows a page created simply for the purpose of display a sentence. The **`Page_Load`** event for this page includes the statement "**`Trace.Warn( "Using warnings" );`**". However, as you see the **`Trace.Warn`** statement did not execute.


http://localhost/TraceProject/WebForm1.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/TraceProject/WebForm1.aspx
Go Links
Done
Local intranet

This is a page that does not do anything.

Fig. 20.22 A page with tracing turned off.


http://localhost/TraceProject/WebForm1.aspx - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/TraceProject/WebForm1.aspx
Go Links
Done
Local intranet

This is a page that does not do anything.

Request Details

Session Id:	x4cbe4f4os0ead55vl5n5jjn	**Request Type:**	GET
Time of Request:	7/17/2001 10:30:40 AM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
	Using warnings.	0.001548	0.001548
aspx.page	End Init	0.001656	0.000107
aspx.page	Begin PreRender	0.002078	0.000423
aspx.page	End PreRender	0.002155	0.000077
aspx.page	Begin SaveViewState	0.002527	0.000372
aspx.page	End SaveViewState	0.003291	0.000765
aspx.page	Begin Render	0.003362	0.000071
aspx.page	End Render	0.038256	0.034894

Control Tree

Control Id	Type	Render Size Bytes (including children)	Viewstate Size Bytes (excluding children)
__PAGE	ASP.WebForm1_aspx	664	24
ctrl0	System.Web.UI.ResourceBasedLiteralControl	431	0
Form1	System.Web.UI.HtmlControls.HtmlForm	161	0
ctrl1	System.Web.UI.LiteralControl	8	0
ctrl2	System.Web.UI.LiteralControl	72	0

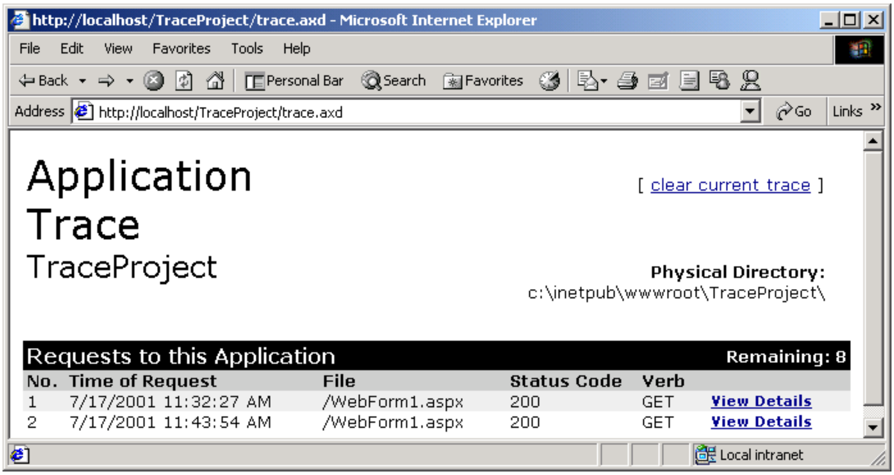
Fig. 20.23 Viewing a page with tracing enabled.

Figure 20.23 shows a page with the **`trace`** property set to **`True`**. The top of the page contains the original page, below it is the tracing information generated by ASP.NET. The **Trace Information** table contains all the information we wrote using the **`Trace`** object's

`Write` and **`Warn`** methods. Our message is displayed on the second row in red. All messages you print via **`Trace`** will be shown here.

The **Control Tree** table lists all the form variables present on the page. If you scroll down you will see several more tables. The **Cookies Collection** table contains information about the cookies for this project (we will talk more about cookies shortly), the **Headers Collection** table contains the HTTP headers for the page and the **Server Variables** table contains a list of server variables and their values.

Tracing is also available for the entire project. To turn on application level tracing open the ***`Web.config`*** file for your project. **`Web.config`** contains information about your project in XML format. Set the ***`enabled`*** property to **`true`** in the ***`trace`*** element. To view the tracing information about the project navigate your browser to the ***`trace.axd`*** file in your project folder. This file does not actually exist on your hard drive; instead it is generated by ASP.NET when you type in the internet address. Figure 20.24 shows the Web page generated by viewing the **`trace.axd`** file.


http://localhost/TraceProject/trace.axd - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address http://localhost/TraceProject/trace.axd
Go Links
Local intranet

Application Trace

TraceProject

[clear current trace]

Physical Directory:
c:\inetpub\wwwroot\TraceProject\

Requests to this Application **Remaining: 8**

No.	Time of Request	File	Status Code	Verb	
1	7/17/2001 11:32:27 AM	/WebForm1.aspx	200	GET	View Details
2	7/17/2001 11:43:54 AM	/WebForm1.aspx	200	GET	View Details

Fig. 20.24 Viewing the tracing information for a project.

This page lists all the pages that were accessed for the project and when they were accessed. Clicking on one of the **View Details** links will take the browser to a page similar to the one in Fig. 20.23. The page for which the tracing information is for, will not be displayed at the top as it was in Fig. 20.23.

We now move on to another feature of ASP.NET—*cookieless session state*. *Cookies* are files stored on your computer by Web sites when you visit one of their Web pages. These Web sites store in those files information pertinent to you—a site may for example want to be able to log you in from your computer automatically. It will store your log in name and password in a cookie and when you next visit the site it will check it—if it contains a log in name and password, it will log you in automatically. Cookies are used frequently by commercial Web sites for many different purposes; an example of using cookies in a shopping cart application is provided in Chapter 23.

Cookies are very useful, the only catch is that this allows Web sites to write to clients' computers. Some people do not like giving this kind of access to Web sites and thus set their

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

browsers to reject cookies from all Web sites. This poses a problem because it is often necessary for Web applications to use cookies to function properly and efficiently. ASP.NET solves this problem by providing a cookieless session state, i.e., simulating cookies without having to write to the client's computer.

To do this, ASP.NET modifies every relative URL in an ASP.NET page by adding to it an ID that uniquely identifies the session. This ID contains all the cookie information for the client that would normally be stored on the client's computer. When this modified URL is received by the server, the ID is decoded to retrieve the cookie information.

By default, an application will use cookies. To have an application take advantage of this functionality, the ***`cookieless`*** property of the ***`sessionState`*** tag in **`Web.config`** must be set to **`true`**. We provide a detailed example of using cookies in Chapter 19, Security.

SUMMARY

- ASP.NET is a technology that allows for a simpler and quicker way to create Web applications.
- To create an ASP.NET Web Form project create a project of type **Web Application**.
- A Web Form is a Web version of Windows Form. It will always have the extension **`.aspx`** and is there to contain all the visual components of your page—HTML code.
- To add controls such as buttons or textboxes to the page they can be dragged from the **Toolbox** to the Web Form.
- All controls, as was the case with Windows Form, are objects, including the page itself. This means they may have properties, methods and events. The properties and events can be changed and set just like before in the **Properties** window.
- These Web Form controls are in the **`System.Web.UI.WebControls`** namespace, not in the **`System.Windows.Forms`** namespace. Even though some classes such as the **`TextBox`** class belong in both namespaces, they are in fact two completely different classes. One is designed to be used on Windows Forms, the other on ASP.NET pages.
- By default the **`pageLayout`** property of the page will be in **`GridLayout`** mode by default, meaning all controls will be placed on a grid and assigned x and y coordinates—their placement does not depend on one another. This can be changed by changing the **`pageLayout`** property of **`Document`** to **`FlowLayout`**—all controls are sequentially lined up as soon as they are placed.
- Every ASP.NET page will have a corresponding C# class which can contain event handlers, initialization code and anything else that the page may need. The file in which this class is contained is called the code-behind file.
- Every ASP.NET page is actually an object of type **`Page`**. Among others, **`Page`** has three properties: **`Request`** of type **`HttpRequest`**, **`Response`** of type **`HttpResponse`**, and **`Session`** of type **`HttpSessionState`**.
- **`Request`** provides information about the incoming HTTP request such as the values of the variables in the form.
- **`Response`** sends information back to the clients browser.
- **`Session`** provides information about the current session. It stores session variables—variables that can be accessed by any page during the same session.
- The "**`<%`**" and "**`%>`**" tags indicate that whatever is in between these constitutes a client-side script. A script is a set of instructions for another program, not a CPU, to execute. A client-side script is a script that executes on the clients computer on their Web browser.

- Web pages often contain ads by corporate sponsers advertising their service or products. They may only have one or two spots on their page for ads, and want to be able to cycle through all the different ones they have easily. The **`AdRotator`** class is designed for such a purpose.
- The **`AdRotator`** class uses information in an XML file it can randomly choose an image to display, set it to link to the appropriate page and show the appropriate caption. If the browser does not support images it will show the alternate text as directed by the XML file.
- The XML file the **`AdRotator`** uses for its image information contains several **`Ad`** nodes, each of which contain a complete set of information about each ad.
- A validator is a control which checks that the data in another control is valid and consistent.
- An object of type **`LiteralControl`** is a control that is usually shown as plain text on a page.
- An object of type **`HtmlGenericControl`** represents any HTML control.
- A server side include (SSI) statement incorporates the contents of a file into an ASPX file. Server-side includes are commands embedded in HTML documents that add dynamic content. The SSI statement is replaced with the contents of the file specified. Not all Web servers support the available SSI commands. Therefore, SSI commands are written as HTML comments. SSI statements always execute before any scripting code executes.
- **`HttpSessionState`**’s method **`Add`** adds session variables to the current session. Its indexer can be used to retrieve them, modify them or check if they are already there.
- **`HttpResponse`**’s method **`Write`** can be used to output HTML code to the browser.
- **`HttpRequest`**’s **`Form`** property contains the information about the contents of the referring page’s form variables.
- Tracing can be enabled for an individual page by setting the **`trace`** property to **`True`**.
- Tracing can be enabled for a project by setting the **`enabled`** property in the **`trace`** tag in **`Web.config`** to **`true`**.
- ASP.NET allows programmer to create programs with cookieless session state for those clients who do not want Web sites to deposit cookies on their computers.

TERMINOLOGY

%> tag
<% tag
Active Server Pages
ActiveX component
`Ad` attribute in XML file
`Add` method of class **`HttpSessionState`**
`ADODataSetCommand` class
`AdRotator` class
`AdvertisementFile` property of class **`AdRotator`**
`AlternateText` attribute in XML file
application tracing
ASP
ASP.NET
ASP.NET page
ASP.NET Web Application project
aspx file extension
`AutoEventWireup` attribute of ASP.NET pages
`Button` class of namespace **`System.Web.UI.WebControls`**
caching Web pages

CheckBox class of namespace **System.Web.UI.WebControls**
CheckBoxList class
client-side script
code-behind file
CommandText property of class **ADOCommand**
CompareValidator class
ControlToValidate property of class **RegularExpressionValidator**
cookie
cookieless property of **sessionState** element in **Web.config**
cookieless session state
CustomValidator class
debugging
discovery file for ASP.NET
DNS lookup
Document property of ASP.NET pages
domain name server (DNS)
DropDownList class
enabled property of **trace** element in **Web.config**
ErrorMessage property of class **RegularExpressionValidator**
FileStream class
FlowLayout mode of **pageLayout** property of ASP.NET pages
GridLayout mode of **pageLayout** property of ASP.NET pages
host
hostname
HTML (HyperText Markup Language)
HTML code
HTML tag
HtmlGenericControl class
HTTP (HyperText Transfer Protocol)
HTTP header
HTTP method
HTTP request type
HttpRequest class
HttpResponse class
HttpSessionState class
HyperLink class
HyperText Markup Language (HTML)
HyperText Transfer Protocol (HTTP)
IIS Web Server
image/gif MIME type
ImageUrl attribute in XML file
Impressions attribute in XML file
Indexer of class **HttpSessionState**
Inherits attribute of ASP.NET pages
Init event of class **Page**
InitializeComponent
instant page content builder
IOException class
IP address
LinkButton class

LiteralControl class
Load even of class **Page**
markup
MIME type
MIME type **image/gif**
MIME type **text/html**
MIME type **text/txt**
NavigateUrl attribute in XML file
NavigateUrl property of class **HyperLink**
.NET Framework
online guestbook
Page class
page tracing
pageLayout property of ASP.NET pages
path to a resource
physical path on the server
postback
processing directives in ASP.NET pages
Properties window
QueryString property of class **HttpRequest**
RadioButtonList class
RangeValidator class
ReadToEnd method of class **StreamReader**
RegularExpressionValidator class
request method
Request property of class**Page**
RequiredFieldValidator class
Response property of class **Page**
scripting engine
SelectCommand property of class **ADODataSetCommand**
server root directory
server-side include (SSI)
Session property of class **Page**
session variable
sessionState element of **Web.config**
.sln solution file
SSI statement
static documents
StreamReader class
StreamWriter class
System.Web.UI
System.Web.UI.WebControls namespace
System.Windows.Forms namespace
Table class of namespace **System.Web.UI.WebControls**
TableCell class
TableRow class
tag
Text property of class **HyperLink**
text/html MIME type
text/txt MIME type

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

TextBox class of namespace **System.Web.UI.WebControls**
title HTML element (**<title>…</title>**)
Toolbox
trace element in **Web.config**
trace property of an ASP.NET page
Trace property of class **Page**
trace.axd file
TraceContext class
tracing
URL (Universal Resource Locator)
validating information
ValidationExpression property of class **RegularExpressionValidator**
validator
virtual directory
virtual in the SSI
virtual path
vsdisco file extension
Warn method of class **TraceContext**
Web Form
Web.config file
Write method of class **HttpResponse**
Write method of class **TraceContext**

SELF-REVIEW EXERCISES

20.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) **FlowLayout** is the default setting of the **pageLayout** property.
b) It is possible to enable tracing in an individual page or in an entire application in ASP.NET.
c) ASP.NET file names typically end in **.aspx**.
d) There is no limit on the number of session variables that may be used.
e) No more than two validator controls may be placed on any control.
f) The **TextBox** control we use in Web Forms is not the same **TextBox** control we use in Windows Forms.
g) An **AdRotator** displays all ads with equal frequency.
h) The file which contains image information for an **AdRotator** may be in formats other than XML.
i) **HttpResponse** method **Redirect** can only be used to redirect the browser to an ASP.NET page within the same folder.
j) Changes made to properties of controls in the **Properties** window are reflected in the **InitializeComponent** method in the code-behind file.
k) If a Web Application Project is not compiled before viewing the page(s) the page(s) will not be displayed.

20.2 Fill in the blanks in each of the following:

a) HTML documents are ________ , while ASP.NET pages provide ________ dynamic Web content to users.
b) A control that ensures that the data in another control is in a specific format is called a __________.
c) Code embedded in an ASP.NET page which executes on the client's computer when the page is loaded is called a _________.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

d) Every ASP.NET page is an instance of the ________ class.

e) When a page loads the __________ event occurs first and the _________ event occurs afterwards.

f) The code necessary for an ASP.NET page is contained in the __________ file.

g) Method _________ of the ________ property of class **`Page`** is used to output HTML code to a page.

h) An HTML page is loaded due to a HTTP _________ request.

i) To programmatically fill a **`Table`** at runtime controls need to be placed in a _________, which gets placed in a _________, which in turn gets placed in the **`Table`**.

j) **`AdRotator`** __________ property points to the file which contains information about all the ads to be displayed in ________ format.

k) The _______ property in the Web Form designer allows us to create controls by either lining them up or by placing them on a grid.

l) Code generated by Visual Studio during the design of an ASP.NET page is placed in the __________ method.

m) If a Web Application Project is not compiled before viewing the page(s) the page(s) will not be displayed.

ANSWERS TO SELF-REVIEW EXERCISES

20.1 a) False. **`GridLayout`** is the default setting of the **`pageLayout`** property. b) True. c) True. d) True. e)False. An unlimited number of validation controls may be placed on one control. f) True. g) False. The **`AdRotator`** displays the ads with frequencies as specified in the **`AdvertisementFile`**. h) False. The **`AdvertisementFile`** may only be an XML file. i) Flase. **`Redirect`** may be used to redirect the user to any page. j) False. Changes to properties of controls can be seen in the ASPX file. k) True.

20.2 a) static, dynamic. b) validator. c) client-side script. d) **`Page`**. e) **`Init`**, **`Load`**. f) code-behind. g) **`Write`**, **`Response`**. h) **`GET`**. i) **`TableCell`**, **`TableRow`**. j) **`AdvertimesementFile`**. k) **`pageLayout`** l) **`InitializeComponent`**.

EXERCISES

20.3 Modify the first example to allow users to select their time zones and then display the time for that time zone. The time should be updated every second.

20.4 Modify the first example to contain drop-down lists for **`Label`** properties such as background color, fore color, font, etc.. Allow the user to select from these lists and then reload the page with the appropriate changes made to the properties of the **`Label`** displaying the time.

20.5 Create an ASP.NET page which uses a file on disk to keep track of how many hits it has received. Display this number every time the page loads.

20.6 Modify the Guestbook case study in Section 20.6.1 to read and write to a database rather than a text file.

20.7 Using the same techniques as the Guestbook case study in Section 20.6.1 develop an ASP.NET application for a discussion group. Allow new links to be created for new topics.

20.8 Create a set of ASP.NET pages which allows users to manipulate a database. Create a database for a book seller with the following fields: BookName, Price, Quantity. The main ASP.NET page should allow users to select from a drop-down list which will contain options to enter more information into the database, view the entire database, update a row from the database, and delete an item from the database. After the user completes an operation they should be able to return to the main page via a link or redirection to begin another operation.

[***Notes To Reviewers***]

Questions:

- Page Builder example is still in Beta1. I am having trouble with the "action" attribute of the form. I am unable to set it, whatever changes I make in the ASPX file do not get reflected when the page is loaded in IE - the attribute gets overwritten. Please help, we have not been able to get much feedback on this.

Comments:

- Please mark your comments in place on a paper copy of the chapter.
- Please return only marked pages to Deitel & Associates, Inc.
- Please do not send us e-mails with detailed, line-by-line comments; mark these directly on the paper pages.
- Please feel free to send any lengthy additional comments by e-mail to **`cheryl.yaeger@deitel.net`**.
- Please run all the code examples.
- Please check that we are using the correct programming idioms.
- Please check that there are no inconsistencies, errors or omissions in the chapter discussions.
- The manuscript is being copyedited by a professional copy editor in parallel with your reviews. That person will probably find most typos, spelling errors, grammatical errors, etc.
- Please do not rewrite the manuscript. We are concerned mostly with technical correctness and correct use of idiom. We will not make significant adjustments to our writing style on a global scale. Please send us a short e-mail if you would like to make such a suggestion.
- Please be constructive. This book will be published soon. We all want to publish the best possible book.
- If you find something that is incorrect, please show us how to correct it.
- Please read all of the back matter including the exercises and any solutions we provide.
- Please review the index we provide with each chapter to be sure we have covered the topics you feel are important.

Symbols

%> tag 975
<% tag 975

A

Active Server Pages 969
ActiveX component 968
Ad attribute in XML file 980
Add method of class **HttpSessionState** 999
ADODataSetCommand class 1001
AdRotator class 978, 979
AdvertisementFile property of class **AdRotator** 979
AlternateText attribute in XML file 980
application tracing 1016
ASP 969
ASP.NET 969, 972, 1016
ASP.NET page 973, 975
ASP.NET Web Application project 972
aspx file extension 973
AutoEventWireup attribute of ASP.NET pages 975

B

Button class of namespace **System.Web.UI.WebControls** 981

C

caching Web pages 969
CheckBox class of namespace **System.Web.UI.WebControls** 981
CheckBoxList class 981
client-side script 975, 1010
code-behind file 969, 973
CommandText property of class **ADOCommand** 1003
CompareValidator class 981
ControlToValidate property of class **RegularExpressionValidator** 983
cookie 1018
cookieless property of **sessionState** element in **Web.config** 1019
cookieless session state 1018
CustomValidator class 982

D

debugging 1016
discovery file for ASP.NET 969
DNS lookup 970
Document property of ASP.NET pages 973
domain name server (DNS) 970
DropDownList class 999

E

enabled property of **trace** element in **Web.config** 1018
ErrorMessage property of class **RegularExpressionValidator** 983

F

FileStream class 996
FlowLayout mode of **pageLayout** property of ASP.NET pages 973

G

GridLayout mode of **pageLayout** property of ASP.NET pages 973, 1019

H

host 970
hostname 970
HTML (HyperText Markup Language) 970
HTML code 975
HTML tag 970
HtmlGenericControl class 997
HTTP (HyperText Transfer Protocol) 970
HTTP header 972
HTTP method 971
HTTP request type 969
HttpRequest class 973
HttpResponse class 973
HttpSessionState class 973
HyperLink class 981
HyperText Markup Language (HTML) 970
HyperText Transfer Protocol (HTTP) 970

I

IIS Web Server 972
image/gif MIME type 972
ImageUrl attribute in XML file 980
Impressions attribute in XML file 980
Indexer of class **HttpSessionState** 999
Inherits attribute of ASP.NET pages 975, 1015
Init event of class **Page** 975, 977
InitializeComponent 977
instant page content builder 997
IOException class 997
IP address 970

L

LinkButton class 981
LiteralControl class 996
Load even of class **Page** 977

M

markup 970
MIME type 972
MIME type **image/gif** 972
MIME type **text/html** 972
MIME type **text/txt** 972

N

NavigateUrl attribute in XML file 980
NavigateUrl property of class **HyperLink** 997
.NET Framework 969

O

online guestbook 988

P

Page class 973
page tracing 1016
pageLayout property of ASP.NET pages 973
path to a resource 970
physical path on the server 999
postback 987
processing directives in ASP.NET pages 975
Properties window 973

Q

QueryString property of class **HttpRequest** 1005

R

RadioButtonList class 981
RangeValidator class 982
ReadToEnd method of class **StreamReader** 996
RegularExpressionValidator class 982
request method 969
Request property of class**Page** 973
RequiredFieldValidator class 981
Response property of class **Page** 973, 999

S

scripting engine 968
SelectCommand property of class **ADODataSetCommand** 1003
server root directory 999
server-side include (SSI) 999
Session property of class **Page** 973
session variable 973, 1005
sessionState element of **Web.config** 1019
.sln solution file 972
SSI statement 999
static documents 968
StreamReader class 996
StreamWriter class 996
System.Web.UI 973
System.Web.UI.WebControls namespace 973
System.Windows.Forms namespace 973

T

Table class of namespace **System.Web.UI.WebControls** 981, 996
TableCell class 996
TableRow class 996
tag 970
Text property of class **HyperLink** 997
text/html MIME type 972
text/txt MIME type 972
TextBox class of namespace **System.Web.UI.WebControls** 981

title HTML element (**<title>…</title>**) 970

Toolbox 973

trace element in **Web.config** 1018

trace property of an ASP.NET page 1016

Trace property of class **Page** 1016

trace.axd file 1018

TraceContext class 1016

tracing 1016

U

URL (Universal Resource Locator) 970

V

validating information 982

ValidationExpression property of class **RegularExpressionValidator** 983

validator 982

virtual directory 970

virtual in the SSI 999

virtual path 999

vsdisco file extension 969

W

Warn method of class **TraceContext** 1016

Web Form 973, 978

Web.config file 1018

Write method of class **HttpResponse** 999, 1013, 1016

Write method of class **TraceContext** 1016

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/17/01

21

ASP .NET and Web Services

Objectives

- To understand what a Web service is.
- To be able to create Web services.
- To understand the elements that comprise a Web service, such as service descriptions and discovery files.
- To be able to create a client that uses a Web service.
- To be able to use Web services with Windows and Web applications.
- To understand session tracking in Web services.
- To be able to pass user-defined data types between Web services and Web clients.

A client is to me a mere unit, a factor in a problem.
Sir Arthur Conan Doyle

...if the simplest things of nature have a message that you understand, rejoice, for your soul is alive.
Eleonora Duse

Protocol is everything.
Francoise Giuliani

They also serve who only stand and wait.
John Milton

Outline

21.1 Introduction
21.2 Web Services
21.3 Simple Object Access Protocol (SOAP) and Web Services
21.4 Publishing and Consuming Web Services
21.5 Session Tracking in Web Services
21.6 Using Web Forms and Web Services
21.7 Case Study: Temperature Information Application
21.8 User-Defined Types in Web Services

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

21.1 Introduction[1]

Throughout this book, we have created dynamic link libraries (DLLs) to facilitate software reusability and modularity—the cornerstones of good object-oriented programming. However, the use of DLLs is limited by the fact that DLLs must reside on the same machine as the programs that use them. This chapter introduces the use of Web services (sometimes called *XML Web services*) to promote software reusability in distributed systems. Distributed-systems technologies allow applications to execute across multiple computers on a network. A Web service is an application that enables distributed computing by allowing one machine to call methods on other machines via common data formats and protocols, such as XML and HTTP. In .NET, these method calls are implemented using the Simple Object Access Protocol (SOAP), an XML-based protocol describing how to mark up requests and responses so that they can be transferred via protocols such as HTTP. Using SOAP, applications represent and transmit data in a standardized format—XML. The underlying implementation of the Web service is irrelevant to clients using the Web service.

Microsoft is encouraging software vendors and e-businesses to deploy Web services. As more and more people worldwide connect to the Internet via networks, applications that call methods across a network becomes more practical. Earlier in this text, we discussed the merits of object-oriented programming. Web services represents the next step in object-oriented programming: Instead of developing software from a small number of class libraries provided at one location, programmers can access countless libraries in multiple locations.

This technology also makes it easier for businesses to collaborate and grow together. By purchasing Web services that are relevant to their businesses, companies that create applications can spend less time coding and more time developing new products from existing components. In addition, e-businesses can employ Web services to provide their customers with an enhanced shopping experience. As a simple example, consider an online music store that enables users to purchase music CDs or to obtain information about artists. Now, suppose another company that sells concert tickets provides a Web service that determines the dates of upcoming concerts by various artists and allows users to buy concert tickets. By licensing the concert-ticket Web service for use on its site, the online music store can sell concert tickets to its customers, which likely will result in increased traffic to its site. The company that sells concert tickets also benefits from the business relationship. In addition to selling more tickets, the company receives revenue from the online music store in exchange for the use of its Web service.

---

[1] Internet Information Services (IIS) must be running to create a Web service in Visual Studio.

Visual Studio and the .NET Framework provide a simple way to create Web services like the one discussed in this example. In this chapter, we explore the steps involved in both the creation and accessing of Web services. For each example, we provide the code for the Web service, then give an example of an application that might use the Web service. Our initial examples are designed to offer a brief introduction to Web services and how they work in Visual Studio. In later sections, we move on to demonstrate more sophisticated Web services.

21.2 Web Services

A Web service is an application stored on one machine that can be accessed on another machine over a network. Due to the nature of this relationship, the machine on which the Web service resides commonly is referred to as a *remote machine*. The application that accesses the Web service sends a method call to the remote machine, which processes the call and sends a response to the application. This kind of distributed computing benefits various systems, including those without access to certain data and those lacking the processing power necessary to perform specific computations.

A Web service is, in its simplest form, a class. In previous chapters, when we wanted to include a class in a project, we would either define the class in our project or add a reference to the compiled DLL. This compiled DLL is placed in the **`bin`** directory of an application by default. As a result, all pieces of our application reside on one machine. When using Web services, the class (and its compiled DLL) we wish to include in our project are stored on a remote machine—a compiled version of this class is not placed in the current application.

Methods in a Web service are remotely invoked using a *Remote Procedure Call* (*RPC*). These methods, which are marked with the ***`WebMethod`*** attribute, often are referred to as *Web-service methods*. Declaring a method with this attribute makes the method accessible to other classes via an RPC. The declaration of a Web-service method with attribute **`WebMethod`** is known as *exposing* the method, or enabling it to be called remotely.

Common Programming Error 21.1

Attempting to call a remote method from a Web service if the method is not declared with the ***`WebMethod`*** *attribute is a compilation error.*

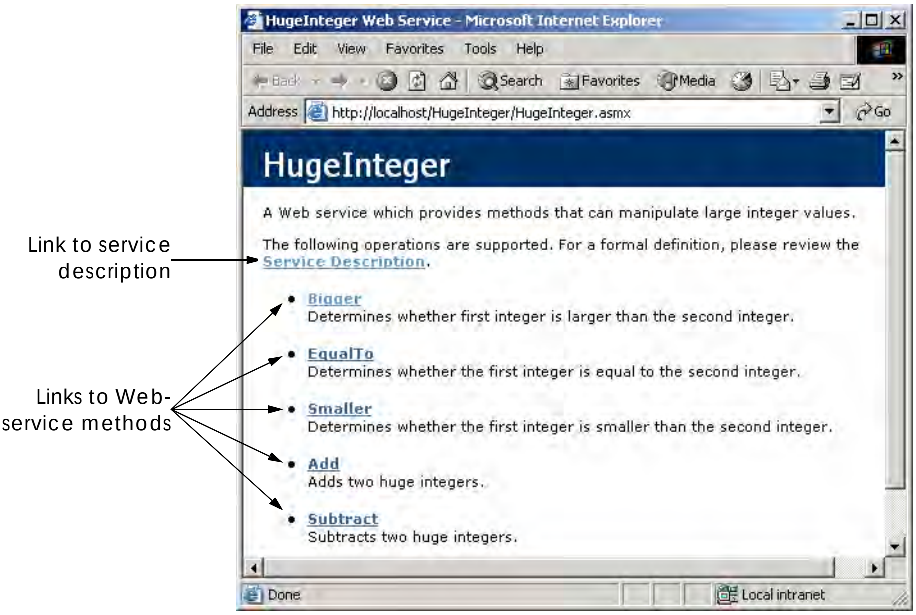
Most requests to and responses from Web services are transmitted via SOAP. This means that any client capable of generating and processing SOAP messages can use a Web service, regardless of the language in which the Web service is written.

Web services have important implications for *business-to-business* (*B2B*) *transactions*, (i.e., transactions that occur between two or more businesses). Now, instead of using proprietary applications, businesses can conduct transactions via Web services—a much simpler and more efficient means of conducting business. Because Web services and SOAP are platform-independent, companies can collaborate and use Web services without worrying about the compatibility of various technologies or programming languages. In this way, Web services are an inexpensive, readily-available solution to facilitate B2B transactions.

A Web service created in Visual Studio .NET has two parts: An *ASMX* file and a code-behind file. The ASMX file by default can be viewed in any Web browser and contains valuable information about the Web service, such as descriptions of Web-service methods and ways to test these methods. The code-behind file provides the implementation for the methods that the Web service encompasses. Figure 21.1 depicts Internet Explorer rendering an ASMX file.

The top of the page provides a link to the Web service's ***Service Description***. A service description is an XML document that conforms to the *Web Service Description Language* (*WSDL*), an XML vocabulary that defines the methods that the Web service makes available and the ways in which clients can interact with those methods. The WSDL document also specifies lower-level information that clients might need, such as the required formats for requests and responses. Visual Studio .NET generates the WSDL service description. Client programs can use the service description to confirm the correctness of method calls when the client programs are compiled.

The programmer should not alter the service description, as it defines how a Web service works. When a user clicks the **Service Description** link at the top of the ASMX page, WSDL is displayed that defines the service description for this Web service (Fig. 21.2).


HugeInteger Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/HugeInteger/HugeInteger.asmx
Go
Link to service description
Links to Web-service methods
Done
Local intranet

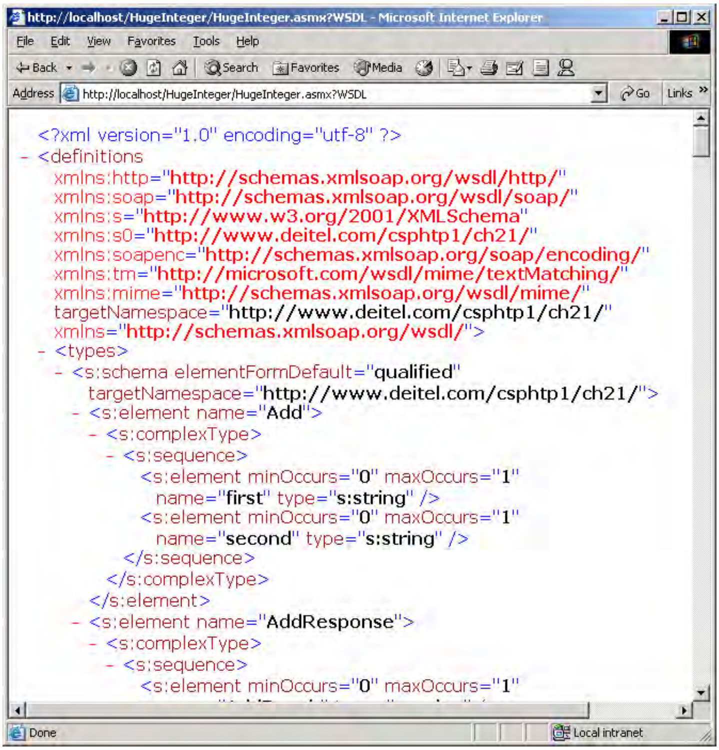
HugeInteger

A Web service which provides methods that can manipulate large integer values.

The following operations are supported. For a formal definition, please review the Service Description.

- Bigger
  Determines whether first integer is larger than the second integer.
- EqualTo
  Determines whether the first integer is equal to the second integer.
- Smaller
  Determines whether the first integer is smaller than the second integer.
- Add
  Adds two huge integers.
- Subtract
  Subtracts two huge integers.

Fig. 21.1 ASMX file rendered in Internet Explorer.

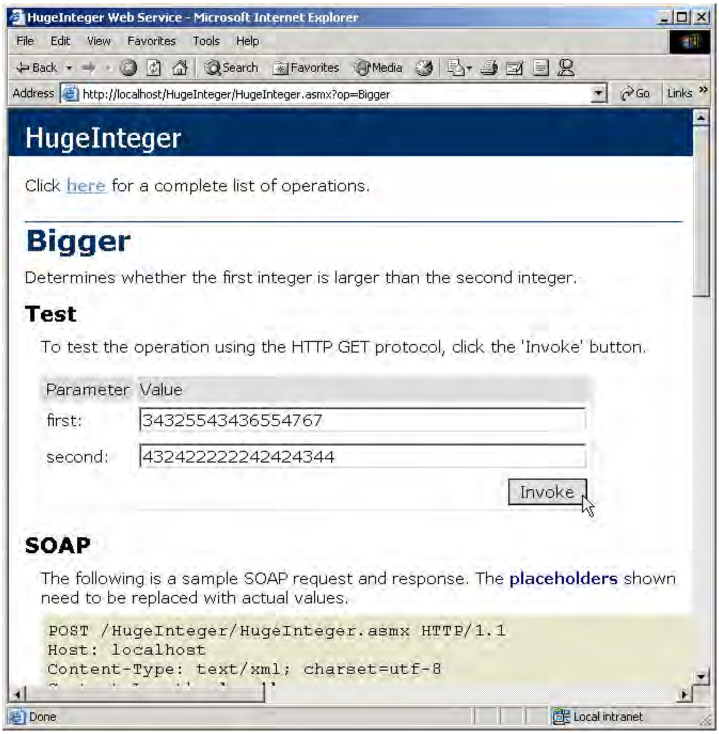

http://localhost/HugeInteger/HugeInteger.asmx?WSDL - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/HugeInteger/HugeInteger.asmx?WSDL
Go Links
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <definitions
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:s0="http://www.deitel.com/csphtp1/ch21/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace="http://www.deitel.com/csphtp1/ch21/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <s:schema elementFormDefault="qualified"
        targetNamespace="http://www.deitel.com/csphtp1/ch21/">
      - <s:element name="Add">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="0" maxOccurs="1"
                name="first" type="s:string" />
              <s:element minOccurs="0" maxOccurs="1"
                name="second" type="s:string" />
            </s:sequence>
          </s:complexType>
        </s:element>
      - <s:element name="AddResponse">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="0" maxOccurs="1"
```

Fig. 21.2 Service description for a Web service.

Below the **Service Description** link, the Web page shown in Fig. 21.1 lists the methods that the Web service provides (i.e., all methods in the application that are declared with **`WebMethod`** attributes). Clicking any method name requests a test page that describes the method (Fig. 21.3). After explaining the method's arguments, the test page allows users to test the method by entering the proper parameters and clicking **Invoke**. (We discuss the process of testing a Web-service method shortly.) Below the **Invoke** button, the page displays sample request and response messages using SOAP, HTTP GET and HTTP POST. These protocols are the three options for sending and receiving messages in Web services. The protocol used to transmit request and response messages is sometimes known as the Web service's *wire protocol* or *wire format*, because the protocol specifies how information is sent "along the wire." Notice that Fig. 21.3 uses the HTTP GET protocol to test a method. Later in this chapter, when we use Web services in our C# programs, we use SOAP as the wire protocol. The advantages to using SOAP over HTTP GET and HTTP POST are discussed in the next section.

On the page depicted in Fig. 21.3, users can test a method by entering **Value**s in the **first:** and **second:** fields and then clicking **Invoke** (in this example, we tested method **`Bigger`**). The method executes, and a new Web browser window opens to display an XML document containing the result (Fig. 21.4). Now that we have introduced a simple example using a Web service, the next several sections explore the role of XML in Web services, as well as other aspects of Web service functionality.


HugeInteger Web Service - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Media
Address
http://localhost/HugeInteger/HugeInteger.asmx?op=Bigger
Go
Links
Invoke
Done
Local intranet

HugeInteger

Click here for a complete list of operations.

Bigger

Determines whether the first integer is larger than the second integer.

Test

To test the operation using the HTTP GET protocol, click the 'Invoke' button.

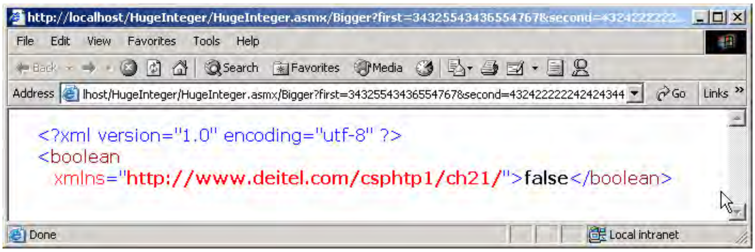
Parameter	Value
first:	34325543436554767
second:	43242222242424344

SOAP

The following is a sample SOAP request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /HugeInteger/HugeInteger.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
```

Fig. 21.3 Invoking a method of a Web service from a Web browser.


http://localhost/HugeInteger/HugeInteger.asmx/Bigger?first=34325543436554767&second=4324222222...
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Media
Address
lhost/HugeInteger/HugeInteger.asmx/Bigger?first=34325543436554767&second=43242222242424344
Go
Links
Done
Local intranet

```
<?xml version="1.0" encoding="utf-8" ?>
<boolean
  xmlns="http://www.deitel.com/csphtp1/ch21/">false</boolean>
```

Fig. 21.4 Results of invoking a Web-service method from a Web browser.

Testing and Debugging Tip 21.1

Using the ASMX page of a Web service to test and debug methods makes that Web service more reliable and robust; it also reduces the likelihood that clients using the Web service will encounter errors.

21.3 Simple Object Access Protocol (SOAP) and Web Services

The Simple Object Access Protocol (SOAP) is a platform-independent protocol that uses XML to make remote-procedure calls over HTTP. Each request and response is packaged in a *SOAP message*—an XML message that contains all the information necessary to process its contents. SOAP messages are quite popular, because they are written in the easy-to-understand and platform-independent XML. Similarly, HTTP was chosen to transmit SOAP messages, because HTTP is a standard protocol for sending information across the Internet. The use of XML and HTTP enables different operating systems to send and receive SOAP messages. Another benefit of HTTP is that it can be used with networks that contain *firewalls*—security barriers that restrict communication among networks.

SOAP supports an extensive set of data types. Readers should note that the wire format used to transmit requests and responses must support all data types passed between the applications. Web services that use SOAP support a wider variety of data types than do Web services that employ other wire formats. The data types supported by SOAP include most basic data types, as well as `DataSet`, `DateTime`, `XmlNode` and several others. SOAP also permits the transmission of arrays of all these types. In addition, user-defined types can be used—we demonstrate how to do this in Section 21.8.

Applications send requests and responses to and from Web services via SOAP. When a program invokes a Web-service method, the request and all relevant information are packaged in a SOAP message and sent to the appropriate destination. When the Web service receives the SOAP message, it begins to process the contents (called the *SOAP envelope*), which specifies the method that the client wishes to execute and the arguments the client is passing to that method. After the Web service receives this request and parses it, the proper method is called with the specified arguments (if there are any), and the response is sent back to the client in another SOAP message. The client parses the response to retrieve the result of the method call.

The SOAP request portrayed in Fig. 21.5 was taken directly from the `Bigger` method of the `HugeInteger` Web service (Fig. 21.3). This Web service provides programmers with several methods that manipulate integers larger than those that can be stored in a `long` variable. Most programmers do not manipulate SOAP messages, allowing the Web service to handle the details of transmission.

Figure 21.5 displays a standard SOAP request that is created when a client wishes to execute the `HugeInteger` Web service's method `Bigger`. When a request to a Web service causes such a SOAP request to be created, the elements `first` and `second`'s character data (`string`s) would contain the actual values that the user entered (lines 16–17). If this envelope contained the request from Fig. 21.3, element `first` and element `second` would contain the values entered in Fig. 21.3. Placeholder `length` would contain the length of this SOAP message.

```
 1  POST /HugeIntegerWebService/HugeInteger.asmx HTTP/1.1
 2  Host: localhost
 3  Content-Type: text/xml; charset=utf-8
 4  Content-Length: length
 5  SOAPAction: "http://www.deitel.com/csphtp1/ch21/Bigger"
 6
 7  <?xml version="1.0" encoding="utf-8"?>
 8
 9  <soap:Envelope
10     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Fig. 21.5 SOAP request for the `HugeInteger` Web service. (Part 1 of 2.)

```
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

    <soap:Body>
       <Bigger xmlns="http://www.deitel.com/csphtp1/ch21/">
          <first>string</first>
          <second>string</second>
       </Bigger>
    </soap:Body>

</soap:Envelope>
```

Fig. 21.5 SOAP request for the **HugeInteger** Web service. (Part 2 of 2.)

21.4 Publishing and Consuming Web Services

This section presents several examples of creating (also known as *publishing*) and using (also known as *consuming*) a Web service. An application that consumes a Web service actually consists of two parts: A *proxy* class that represents the Web service and a client application that accesses the Web service via an instance of the proxy class. The proxy class handles the transferral of the arguments for the Web-service method from the client application to the Web service, as well as the transferral of the result from the Web-service method back to the client application. Visual Studio can generate proxy classes—we demonstrate how to do this momentarily.

Figure 21.6 presents the code-behind file for the **HugeInteger** Web service (Fig. 21.1). The name of the Web service is based on the name of the class that defines it (in this case, **HugeInteger**). This Web service is designed to perform calculations with integers that contain a maximum of 100 digits. As we mentioned earlier, **long** variables cannot handle integers of this size (i.e., an overflow would occur). The Web service provides a client with methods that take two “huge integers” and determine which one is larger or smaller, whether the two numbers are equal, their sum or their difference. The reader can think of these methods as services that one application provides for the programmers of other applications (hence the term, “Web services”). Any programmer can access this Web service, use its methods and thus avoid the writing of over 200 lines of code. In the remaining examples, we hide portions of the Visual Studio generated code in the code-behind files. We do this both for brevity and for presentation purposes.

```
// Fig. 21.6: HugeInteger.asmx.cs
// HugeInteger Web Service.

using System;
using System.Text;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services; // contains Web service related classes
```

Fig. 21.6 **HugeInteger** Web service. (Part 1 of 6.)

```

namespace HugeIntegerWebService
{
   /// <summary>
   /// performs operations on large integers
   /// </summary>
   [ WebService(
      Namespace = "http://www.deitel.com/csphtp1/ch21/",
      Description = "A Web service which provides methods that" +
      " can manipulate large integer values." ) ]
   public class HugeInteger : System.Web.Services.WebService
   {
      // default constructor
      public HugeInteger()
      {
         // CODEGEN: This call is required by the ASP .NET Web
         // Services Designer
         InitializeComponent();

         number = new int[ MAXIMUM ];
      }

      #region Component Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
      }
      #endregion

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose( bool disposing )
      {
      }

      // WEB SERVICE EXAMPLE
      // The HelloWorld() example service returns
      // the string Hello World
      // To build, uncomment the following lines
      // then save and build the project
      // To test this web service, press F5

      //   [WebMethod]
      //   public string HelloWorld()
      //   {
      //      return "Hello World";
      //   }

      private const int MAXIMUM = 100;
```

Fig. 21.6 **`HugeInteger`** Web service. (Part 2 of 6.)

```

      public int[] number;

      // indexer that accepts an integer parameter
      public int this[ int index ]
      {
         get
         {
            return number[ index ];
         }

         set
         {
            number[ index ] = value;
         }

      } // end indexer

      // returns string representation of HugeInteger
      public override string ToString()
      {
         StringBuilder returnString = new StringBuilder();

         foreach ( int digit in number )
            returnString.Insert( 0, digit );

         return returnString.ToString();
      }

      // creates HugeInteger based on argument
      public static HugeInteger FromString( string integer )
      {
         HugeInteger parsedInteger = new HugeInteger();

         for ( int i = 0; i < integer.Length; i++ )
            parsedInteger[ i ] = Int32.Parse(
               integer[ integer.Length - i - 1 ].ToString() );

         return parsedInteger;
      }

      // WebMethod that performs integer addition
      // represented by string arguments
      [ WebMethod ( Description = "Adds two huge integers." ) ]
      public string Add( string first, string second )
      {
         int carry = 0;

         HugeInteger operand1 = HugeInteger.FromString( first );
         HugeInteger operand2 =
            HugeInteger.FromString( second );

```

Fig. 21.6 **`HugeInteger`** Web service. (Part 3 of 6.)

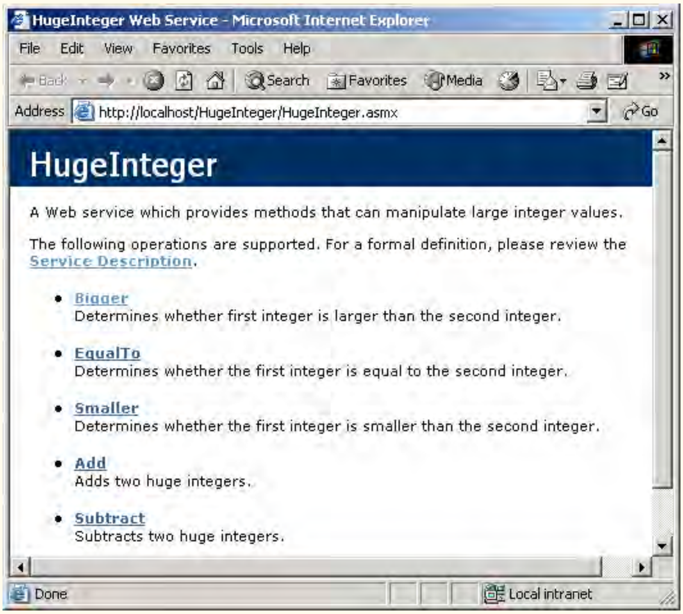
```
         // store result of addition
         HugeInteger result = new HugeInteger();

         // perform addition algorithm for each digit
         for ( int i = 0; i < MAXIMUM; i++ )
         {
            // add two digits in same column
            // result is their sum, plus carry from
            // previous operation modulus 10
            result[ i ] =
               ( operand1[ i ] + operand2[ i ] ) % 10 + carry;

            // store remainder of dividing
            // sums of two digits by 10
            carry = ( operand1[ i ] + operand2[ i ] ) / 10;
         }

         return result.ToString();

      } // end method Add

      // WebMethod that performs the subtraction of integers
      // represented by string arguments
      [ WebMethod (
         Description = "Subtracts two huge integers." ) ]
      public string Subtract( string first, string second )
      {
         HugeInteger operand1 = HugeInteger.FromString( first );
         HugeInteger operand2 =
            HugeInteger.FromString( second );
         HugeInteger result = new HugeInteger();

         // subtract top digit from bottom digit
         for ( int i = 0; i < MAXIMUM; i++ )
         {
            // if top digit is smaller than bottom
            // digit we need to borrow
            if ( operand1[ i ] < operand2[ i ] )
               Borrow( operand1, i );

            // subtract bottom from top
            result[ i ] = operand1[ i ] - operand2[ i ];
         }

         return result.ToString();

      } // end method Subtract

      // borrows 1 from next digit
      private void Borrow( HugeInteger integer, int place )
      {
```

Fig. 21.6 **`HugeInteger`** Web service. (Part 4 of 6.)

```
         // if no place to borrow from, signal problem
         if ( place >= MAXIMUM - 1 )
            throw new ArgumentException();

         // otherwise if next digit is zero,
         // borrow from digit to left
         else if ( integer[ place + 1 ] == 0 )
            Borrow( integer, place + 1 );

         // add ten to current place because we borrowed
         // and subtract one from previous digit -
         // this is digit borrowed from
         integer[ place ] += 10;
         integer[ place + 1 ] -= 1;

      } // end method Borrow

      // WebMethod that returns true if first integer is
      // bigger than second
      [ WebMethod ( Description = "Determines whether first " +
         "integer is larger than the second integer." ) ]
      public bool Bigger( string first, string second )
      {
         char[] zeroes = { '0' };

         try
         {
            // if elimination of all zeroes from result
            // of subtraction is an empty string,
            // numbers are equal, so return false,
            // otherwise return true
            if ( Subtract( first, second ).Trim( zeroes ) == "" )
               return false;
            else
               return true;
         }

         // if ArgumentException occurs, first number
         // was smaller, so return false
         catch ( ArgumentException )
         {
            return false;
         }

      } // end method Bigger

      // WebMethod returns true if first integer is
      // smaller than second
      [ WebMethod ( Description = "Determines whether the " +
         "first integer is smaller than the second integer." ) ]
      public bool Smaller( string first, string second )
      {
```

Fig. 21.6 **`HugeInteger`** Web service. (Part 5 of 6.)

```
220             // if second is bigger than first, then first is
221             // smaller than second
222             return Bigger( second, first );
223          }
224
225          // WebMethod that returns true if two integers are equal
226          [ WebMethod ( Description = "Determines whether the " +
227             "first integer is equal to the second integer." ) ]
228          public bool EqualTo( string first, string second )
229          {
230             // if either first is bigger than second, or first is
231             // smaller than second, they are not equal
232             if ( Bigger( first, second ) ||
233                Smaller( first, second ) )
234                return false;
235             else
236                return true;
237          }
238
239       } // end class HugeInteger
240
241    } // end namespace HugeIntegerWebService
```


HugeInteger Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/HugeInteger/HugeInteger.asmx
Go
Done
Local intranet

HugeInteger

A Web service which provides methods that can manipulate large integer values.

The following operations are supported. For a formal definition, please review the Service Description.

- Bigger
  Determines whether first integer is larger than the second integer.
- EqualTo
  Determines whether the first integer is equal to the second integer.
- Smaller
  Determines whether the first integer is smaller than the second integer.
- Add
  Adds two huge integers.
- Subtract
  Subtracts two huge integers.

Fig. 21.6 **`HugeInteger`** Web service. (Part 6 of 6.)

Line 13 places class **`HugeInteger`** in namespace **`HugeIntegerWebService`**. Line 19 assigns the Web service namespace to **`www.deitel.com/csphtp1/ch21/`** to uniquely identify this Web service. The namespace is specified using the ***`Namespace`*** property of the ***`WebService`*** attribute. In lines 20–21, we use property ***`Description`*** to provide information about our Web service that appears in the ASMX file. Line 22 specifies that our class derives from ***System.Web.Services.WebService***. By default, Visual Studio defines our Web service so that it inherits from the **WebService** class. Although a Web service class is not required to subclass **WebService**, class **WebService** provides members that are useful in determining information about the client and the Web service itself. Several methods in class **HugeInteger** are tagged with the ***WebMethod*** attribute, which *exposes* the method such that it can be called remotely. When this attribute is absent, the method is not accessible through the Web service. Notice that the **WebMethod** attribute, like the **WebService** attribute, contains a **Description** property, which provides information about the method to the ASMX page. Readers can see these descriptions in the output of Fig. 21.6.

Good Programming Practice 21.1

Specify a namespace for each Web service so that it can be uniquely identified.

Good Programming Practice 21.2

Specify descriptions for all Web services and Web-service methods so that clients can obtain additional information about the Web service and its contents.

Common Programming Error 21.2

*Web-service methods cannot be declared **static**, or a runtime error will occur when attempting to view the ASMX page. For a client to access a Web-service method, an instance of that Web service must exist.*

Lines 69–81 define an indexer for our class. This enables us to access any digit in **HugeInteger** as if we were accessing it through array **number**. Lines 108–136 and 142–163 define **WebMethod**s **Add** and **Subtract**, which perform addition and subtraction, respectively. Method **Borrow** (lines 166–183) handles the case in which the digit in the left operand is smaller than the corresponding digit in the right operand. For instance, when we subtract 19 from 32, we usually go digit by digit, starting from the right. The number 2 is smaller than 9, so we add 10 to 2 (resulting in 12), which subtracts 9, resulting in 3 for the rightmost digit in the solution. We then subtract 1 from the next digit over (3), making it 2. The corresponding digit in the right operand is now the “1” in 19. The subtraction of 1 from 2 is 1, making the corresponding digit in the result 1. The final result, when both resulting digits are combined, is 13. Method **Borrow** adds 10 to the appropriate digits and subtracts 1 from the digit to the left. Because this is a utility method that is not intended to be called remotely, it is not qualified with attribute **WebMethod**.

The screen capture in Fig. 21.6 is identical to the one in Fig. 21.1. A client application can invoke only the five methods listed in the screen shot (i.e., the methods qualified with the **WebMethod** attribute).

Now, let us demonstrate how to create this Web service. To begin, we must create a project of type **ASP.NET Web Service**. Like Web Forms, Web services are by default placed in the Web server’s **wwwroot** directory on the server (**localhost**). By default, Visual Studio places the solution file (**.sln**) in the **Visual Studio Projects** folder, in a directory for the solution. (The **Visual Studio Projects** folder is usually located in the **My Documents** folder.)

Notice that, when the project is created, the code-behind file is displayed in design view by default (Fig. 21.7). If this file is not open, it can be opened by clicking

`Service1.asmx`. The file that will be opened, however, is **`Service1.asmx.cs`** (the code-behind file for our Web service). This is because, when creating Web services in Visual Studio, programmers work almost exclusively in the code-behind file. In fact, if a programmer were to open the ASMX file, it would contain only the lines:

```
<%@ WebService Language="c#" Codebehind="Service1.asmx.cs"
    Class="WebService1.Service1" %>
```

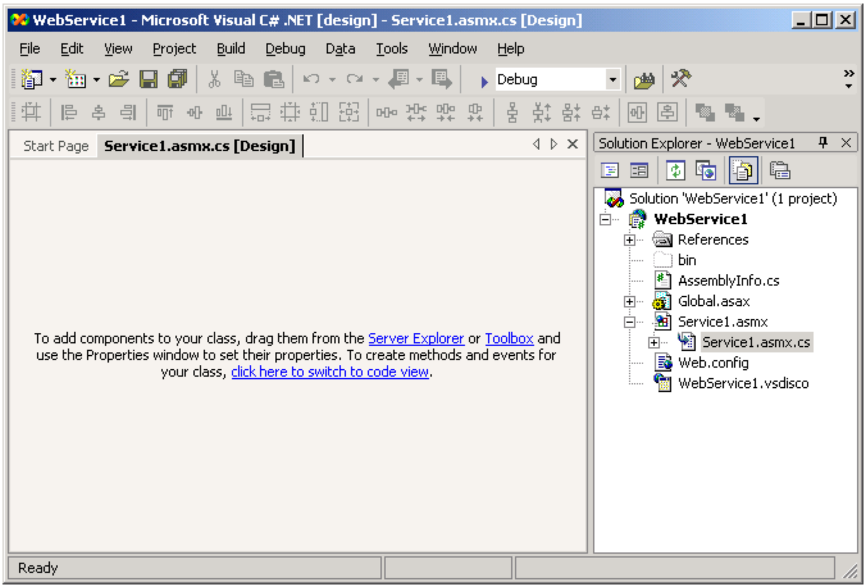
indicating the name of the code-behind file, the programming language in which the code-behind file is written and the class that defines our Web service. This is the extent of the information that this file must contain. [*Note*: By default, the code-behind file is not listed in the **Solution Explorer**. The code-behind file is displayed when the ASMX file is double clicked in the **Solution Explorer**. This file can be listed in the **Solution Explorer** by clicking the icon to show all files.]

It might seem strange that there is a design view for Web services, given that Web services do not have graphical user interfaces. A design view is provided because more sophisticated Web services contain methods that manipulate more than just strings or numbers. For example, a Web-service method could manipulate a database. Instead of typing all the code necessary to create a database connection, developers can simply drop the proper ADO .NET components into the design view and manipulate them as we would in a Windows or Web application. We show an example of this in Section 21.6.

Now that we have defined our Web service, we demonstrate how to use it. First, a client application must be created. In this first example, we create a Windows application as our client. Once this application has been created, the client must add a proxy class for accessing the Web service. A proxy class (or proxy) is a class created from the Web service's WSDL file that enables the client to call Web-service methods over the Internet. The proxy class handles all the “plumbing” required for Web-service method calls. Whenever a call is made in the client application to a Web-service method, the application actually calls a corresponding method in the proxy class. This method takes the name of the method and its arguments, then formats them so that they can be sent as a request in a SOAP message. The Web service receives this request and executes the method call, sending back the result as another SOAP message. When the client application receives the SOAP message containing the response, the proxy class decodes it and formats the results so that they are understandable to the client. This information then is returned to the client. It is important to note that the proxy class essentially is hidden from the programmer. We cannot, in fact, view it in the **Solution Explorer** unless we choose to show all the files. The purpose of the proxy class is to make it seem to clients as though they are calling the Web-service methods directly. It is rarely necessary for the client to view or manipulate the proxy class.

The next example demonstrates how to create a Web service client and its corresponding proxy class. We must begin by creating a project and adding a *Web reference* to that project. When we add a Web reference to a client application, the proxy class is created. The client then creates an instance of the proxy class, which is used to call methods included in the Web service.

To create a proxy in Visual Studio, right click the **References** folder in **Solution Explorer** and select ***Add Web Reference*** (Fig. 21.8). In the **Add Web Reference** dialog that appears (Fig. 21.9), enter the Web address of the Web service and press *Enter*. In this chapter, we store the Web service in the root directory of our local Web server


WebService1 - Microsoft Visual C# .NET [design] - Service1.asmx.cs [Design]
File
Edit
View
Project
Build
Debug
Data
Tools
Window
Help
Debug
Start Page
Service1.asmx.cs [Design]
Solution Explorer - WebService1
Solution 'WebService1' (1 project)
WebService1
References
bin
AssemblyInfo.cs
Global.asax
Service1.asmx
Service1.asmx.cs
Web.config
WebService1.vsdisco
Ready

To add components to your class, drag them from the Server Explorer or Toolbox and use the Properties window to set their properties. To create methods and events for your class, click here to switch to code view.

Fig. 21.7 Design view of a Web service.

(`http://localhost`, whose physical path is `C:\Inetpub\wwwroot`). We now can add a Web reference by clicking the link **Web References on Local Web Server** (Fig. 21.9). Next, we select the appropriate Web service from the list of Web services located on `localhost` (Fig. 21.10). Notice that each Web service is listed as a file with the extension `.vsdisco` that is located in the directory for the Web service project. Files with the extension `.disco` or `.vsdisco` are known as *discovery files*. We discuss discovery files, as well as the distinctions between `.disco` files and `.vsdisco` files, later in this section. Once a Web service is chosen the description of that Web service appears, and the developer can click **Add Reference** (Fig. 21.11). This adds to the **Solution Explorer** (Fig. 21.12) a **Web References** folder with a node named for the domain where the Web service is located. In this case, the name is `localhost`, because we are using the local Web server. This means that, when we reference class `HugeInteger`, we will be doing so through class `HugeInteger` in namespace `localhost`, instead of class `HugeInteger` in namespace `HugeIntegerWebService` [*Note*: The Web service class and the proxy class have the same name. Visual Studio generates a proxy for the Web service and adds it as a reference (Fig. 21.12).]

Good Programming Practice 21.3

When creating a program that will use Web services, add the Web reference first. This will enable Visual Studio to recognize an instance of the Web service class, allowing Intellisense to help the developer use the Web service.

The steps that we described previously work well if the programmer knows the appropriate Web services reference. However, what if we are trying to locate a new Web service? There are two technologies that facilitate this process: *Universal Description, Discovery and Integration* (*UDDI*) and *Discovery files* (*DISCO*). UDDI is a project for developing a set of specifications that define how Web services should be published so that programmers searching for Web services can find them. Microsoft began this ongoing project to facilitate the locating of Web services that conform to certain specifications, allowing programmers to find different Web services using search engines. UDDI organizes and describes Web services and then places this information in a central location. Although UDDI is beyond the scope of what we are teaching, the reader can learn more about this project and view a demonstration by visiting **`www.uddi.org`** and **`uddi.microsoft.com`**. These sites contain search tools that make finding Web services fast and easy.

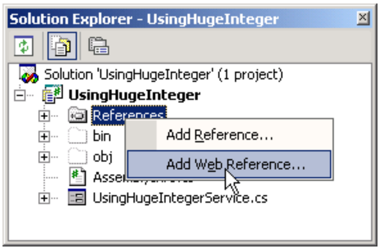
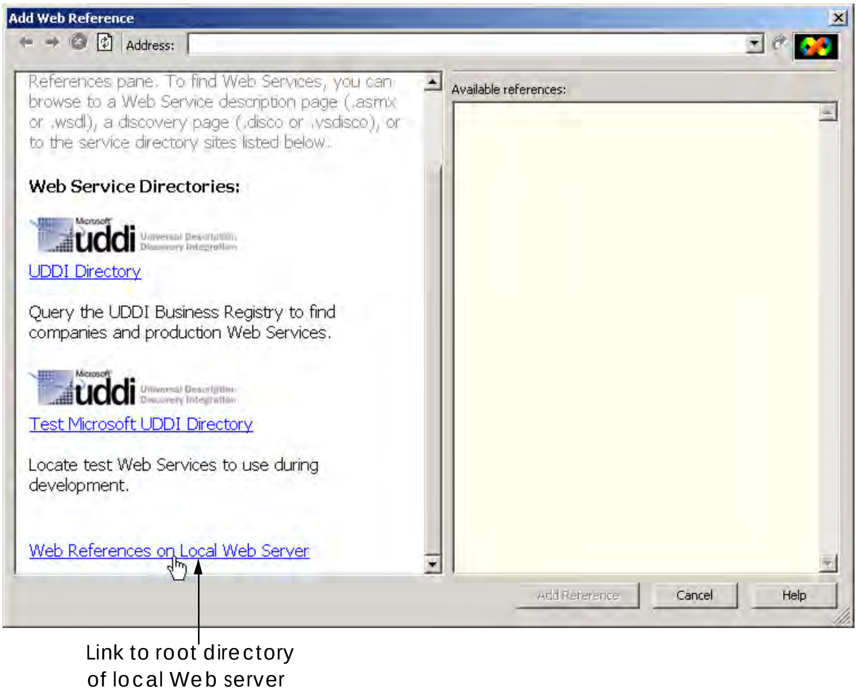

Solution Explorer - UsingHugeInteger
Solution 'UsingHugeInteger' (1 project)
UsingHugeInteger
References
bin
obj
Asser
UsingHugeIntegerService.cs
Add Reference...
Add Web Reference...

Fig. 21.8 Adding a Web service reference to a project.


Add Web Reference
Address:
Available references:
Web Service Directories:
uddi
UDDI Directory
uddi
Test Microsoft UDDI Directory
Web References on Local Web Server
Add Reference
Cancel
Help

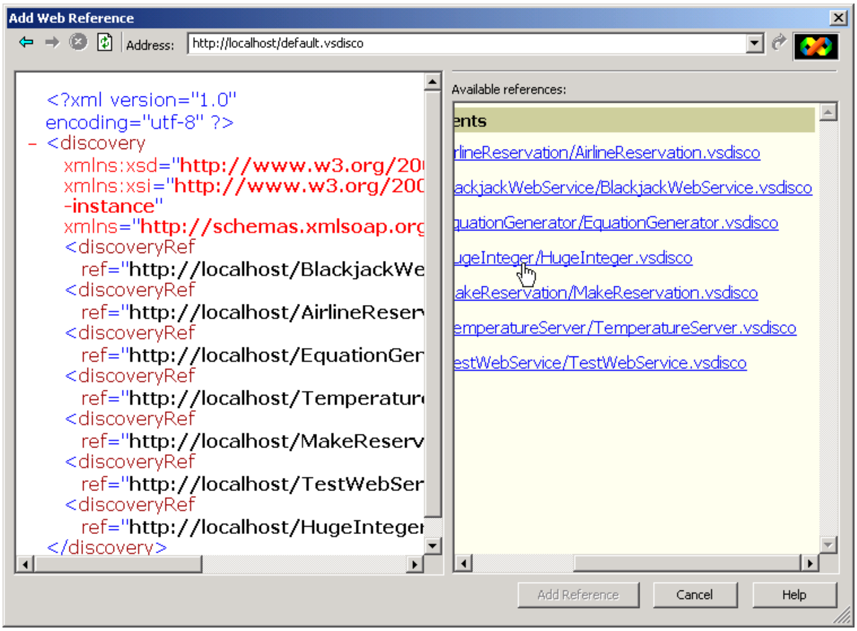
References pane. To find Web Services, you can browse to a Web Service description page (.asmx or .wsdl), a discovery page (.disco or .vsdisco), or to the service directory sites listed below.

Query the UDDI Business Registry to find companies and production Web Services.

Locate test Web Services to use during development.

Link to root directory of local Web server

Fig. 21.9 **Add Web Reference** dialog.



http://localhost/default.vsdisco

```
  <?xml version="1.0"
  encoding="utf-8" ?>
- <discovery
    xmlns:xsd="http://www.w3.org/20
    xmlns:xsi="http://www.w3.org/200
    -instance"
    xmlns="http://schemas.xmlsoap.org
    <discoveryRef
      ref="http://localhost/BlackjackWe
    <discoveryRef
      ref="http://localhost/AirlineReser
    <discoveryRef
      ref="http://localhost/EquationGen
    <discoveryRef
      ref="http://localhost/Temperatur
    <discoveryRef
      ref="http://localhost/MakeReserv
    <discoveryRef
      ref="http://localhost/TestWebSer
    <discoveryRef
      ref="http://localhost/HugeInteger
  </discovery>
```

Available references:

ents

rlineReservation/AirlineReservation.vsdisco

ackjackWebService/BlackjackWebService.vsdisco

quationGenerator/EquationGenerator.vsdisco

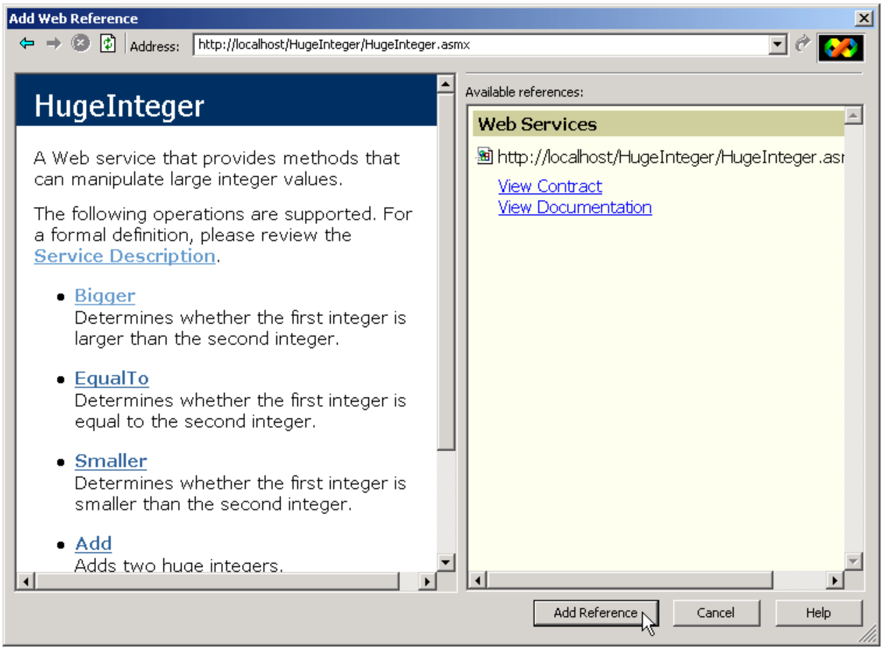
ugeInteger/HugeInteger.vsdisco

akeReservation/MakeReservation.vsdisco

emperatureServer/TemperatureServer.vsdisco

estWebService/TestWebService.vsdisco

Fig. 21.10 Web services located on `localhost`.



http://localhost/HugeInteger/HugeInteger.asmx

HugeInteger

A Web service that provides methods that can manipulate large integer values.

The following operations are supported. For a formal definition, please review the Service Description.

- Bigger
  Determines whether the first integer is larger than the second integer.
- EqualTo
  Determines whether the first integer is equal to the second integer.
- Smaller
  Determines whether the first integer is smaller than the second integer.
- Add
  Adds two huge integers.

Available references:

Web Services

http://localhost/HugeInteger/HugeInteger.asr

View Contract

View Documentation

Fig. 21.11 Web reference selection and description.

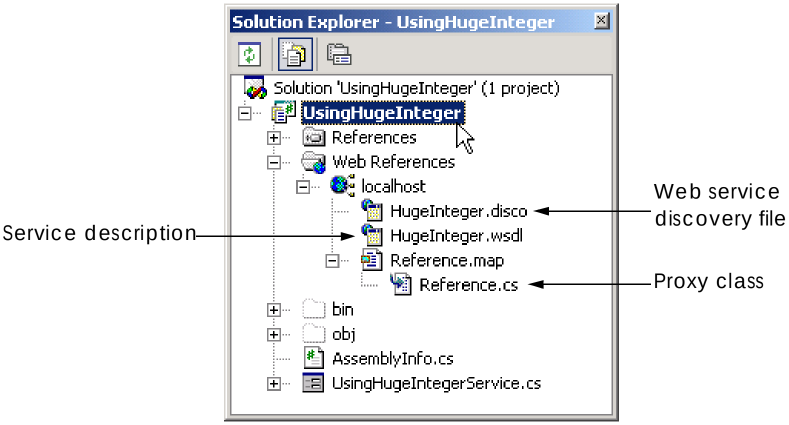

Solution Explorer - UsingHugeInteger
Solution 'UsingHugeInteger' (1 project)
UsingHugeInteger
References
Web References
localhost
HugeInteger.disco
HugeInteger.wsdl
Reference.map
Reference.cs
bin
obj
AssemblyInfo.cs
UsingHugeIntegerService.cs
Web service discovery file
Service description
Proxy class

Fig. 21.12 **Solution Explorer** after adding a Web reference to a project.

A DISCO file catalogs Web services in a particular directory. There are two types of discovery files: *Dynamic discovery* files (with a **`.vsdisco`** extension) and *static discovery* files (with a **`.disco`** extension). These files indicate both the location of the ASMX file and the service description (a WSDL file) for each Web service in the current directory, as well as any Web services in the current directory's subdirectories. When a programmer creates a Web service, Visual Studio generates a dynamic discovery file for that Web service. When a Web reference is added, the client uses the dynamic discovery file to select the desired Web service, as demonstrated in Fig. 21.10. Once the Web reference is created, a static discovery file is placed in the client's project. The static discovery file hard codes the location for the ASMX and WSDL files. (By "hard code", we mean that the location is entered directly into the file.) Dynamic discovery files, on the other hand, are created such that a list of Web services is created dynamically on the server when a client is searching for a Web service. The use of dynamic discovery enables certain extra options, such as hiding of certain Web services in subdirectories. Discovery files are a Microsoft-specific technology, whereas UDDI is not. However, the two can work together to enable a client to find a Web service. Using both technologies, the client can use a search engine to find a location with various Web services on a topic, and then use discovery files to view all the Web services in that location.

Once the Web reference is added, the client can access the Web service through our proxy. Because our proxy class is named **`HugeInteger`** and is located in namespace **`localhost`**, we must use **`localhost.HugeInteger`** to reference this class. The Windows Form in Fig. 21.13 uses the **`HugeInteger`** Web service to perform computations with positive integers up to **`100`** digits long. [*Note*: If using the example on this book's CD, the reader might need to regenerate the proxy.]

```
// Fig. 21.13: UsingHugeIntegerService.cs
// Using the HugeInteger Web service.

```

Fig. 21.13 Using the **`HugeInteger`** Web service. (Part 1 of 6.)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Web.Services.Protocols;

// allows user to perform operations on large integers
public class UsingHugeIntService : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label promptLabel;
   private System.Windows.Forms.Label resultLabel;

   private System.Windows.Forms.TextBox firstTextBox;
   private System.Windows.Forms.TextBox secondTextBox;

   private System.Windows.Forms.Button addButton;
   private System.Windows.Forms.Button subtractButton;
   private System.Windows.Forms.Button biggerButton;
   private System.Windows.Forms.Button smallerButton;
   private System.Windows.Forms.Button equalButton;

   private System.ComponentModel.Container components = null;

   // declare a reference Web service
   private localhost.HugeInteger remoteInteger;

   private char[] zeroes = { '0' };

   // default constructor
   public UsingHugeIntService()
   {
      InitializeComponent();

      // instantiate remoteInteger
      remoteInteger = new localhost.HugeInteger();
   }

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new UsingHugeIntService() );

   } // end Main

   // checks whether two numbers user input are equal
   protected void equalButton_Click(
      object sender, System.EventArgs e )
   {
```

Fig. 21.13 Using the **`HugeInteger`** Web service. (Part 2 of 6.)

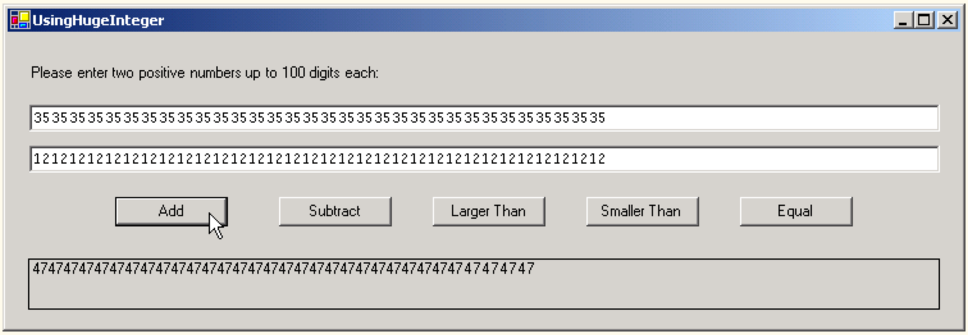
```
         // make sure HugeIntegers do not exceed 100 digits
         if ( CheckSize( firstTextBox, secondTextBox ) )
            return;

         // call Web-service method to determine
         // whether integers are equal
         if ( remoteInteger.EqualTo(
            firstTextBox.Text, secondTextBox.Text ) )

            resultLabel.Text =
               firstTextBox.Text.TrimStart( zeroes ) +
               " is equal to " +
               secondTextBox.Text.TrimStart( zeroes );
         else
            resultLabel.Text =
               firstTextBox.Text.TrimStart( zeroes ) +
               " is NOT equal to " +
               secondTextBox.Text.TrimStart( zeroes );

      } // end method equalButton_Click

      // checks whether first integer input
      // by user is smaller than second
      protected void smallerButton_Click(
         object sender, System.EventArgs e )
      {
         // make sure HugeIntegers do not exceed 100 digits
         if ( CheckSize( firstTextBox, secondTextBox ) )
            return;

         // call Web-service method to determine whether first
         // integer is smaller than second
         if ( remoteInteger.Smaller(
            firstTextBox.Text, secondTextBox.Text ) )

            resultLabel.Text =
               firstTextBox.Text.TrimStart( zeroes ) +
               " is smaller than " +
               secondTextBox.Text.TrimStart( zeroes );
         else
            resultLabel.Text =
               firstTextBox.Text.TrimStart( zeroes ) +
               " is NOT smaller than " +
               secondTextBox.Text.TrimStart( zeroes );

      } // end method smallerButton_Click

      // checks whether first integer input
      // by user is bigger than second
      protected void biggerButton_Click(
         object sender, System.EventArgs e )
      {
```

Fig. 21.13 Using the `HugeInteger` Web service. (Part 3 of 6.)

```
      // make sure HugeIntegers do not exceed 100 digits
      if ( CheckSize( firstTextBox, secondTextBox ) )
         return;

      // call Web-service method to determine whether first
      // integer is larger than the second
      if ( remoteInteger.Bigger( firstTextBox.Text,
         secondTextBox.Text ) )

         resultLabel.Text =
            firstTextBox.Text.TrimStart( zeroes ) +
            " is larger than " +
            secondTextBox.Text.TrimStart( zeroes );
      else
         resultLabel.Text =
            firstTextBox.Text.TrimStart( zeroes ) +
            " is NOT larger than " +
            secondTextBox.Text.TrimStart( zeroes );

   } // end method biggerButton_Click

   // subtract second integer from first
   protected void subtractButton_Click(
      object sender, System.EventArgs e )
   {
      // make sure HugeIntegers do not exceed 100 digits
      if ( CheckSize( firstTextBox, secondTextBox ) )
         return;

      // perform subtraction
      try
      {
         string result = remoteInteger.Subtract(
            firstTextBox.Text,
            secondTextBox.Text ).TrimStart( zeroes );

         resultLabel.Text = ( ( result == "" ) ? "0" : result );
      }

      // if WebMethod throws an exception, then first
      // argument was smaller than second
      catch ( SoapException )
      {
         MessageBox.Show(
            "First argument was smaller than the second" );
      }

   } // end method subtractButton_Click

   // adds two integers input by user
   protected void addButton_Click(
      object sender, System.EventArgs e )
   {
```

Fig. 21.13 Using the **`HugeInteger`** Web service. (Part 4 of 6.)

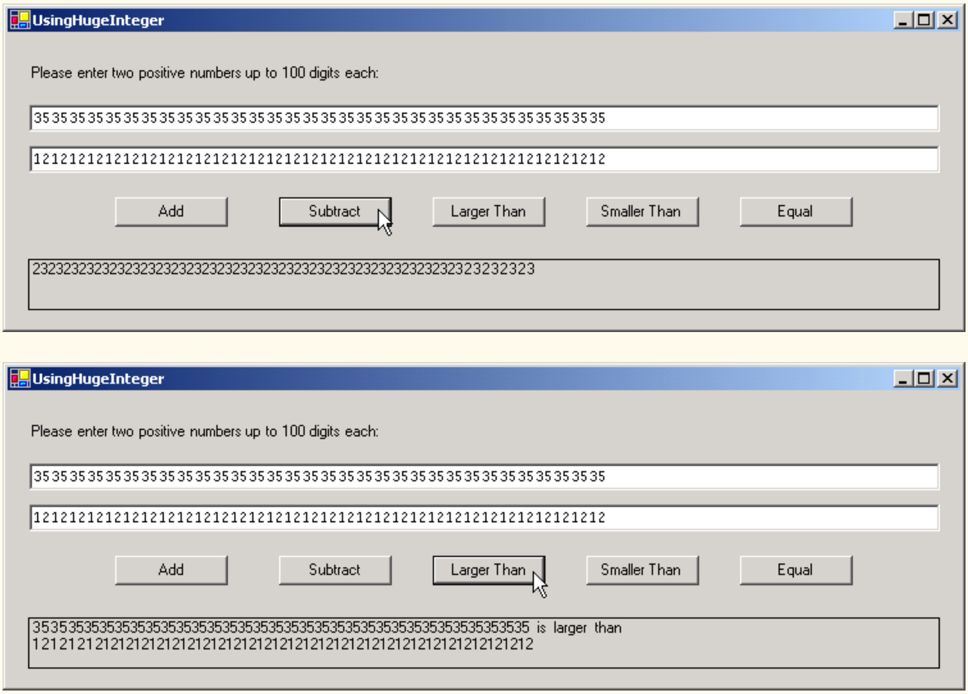
```
      // make sure HugeInteger does not exceed 100 digits
      // and is not situation where both integers are 100
      // digits long--result in overflow
      if ( firstTextBox.Text.Length > 100 ||
         secondTextBox.Text.Length > 100  ||
         ( firstTextBox.Text.Length == 100 &&
         secondTextBox.Text.Length == 100 ) )
      {
         MessageBox.Show( "HugeIntegers must not be more "
            + "than 100 digits\nBoth integers cannot be of"
            + " length 100: this causes an overflow",
            "Error", MessageBoxButtons.OK,
            MessageBoxIcon.Information );

         return;
      }

      // perform addition
      resultLabel.Text = remoteInteger.Add( firstTextBox.Text,
         secondTextBox.Text ).TrimStart( zeroes ).ToString();

   } // end method addButton_Click

   // determines whether size of integers is too big
   private bool CheckSize( TextBox first, TextBox second )
   {
      if ( first.Text.Length > 100 || second.Text.Length > 100 )
      {
         MessageBox.Show( "HugeIntegers must be less than 100"
            + " digits", "Error", MessageBoxButtons.OK,
            MessageBoxIcon.Information );

         return true;
      }

      return false;

   } // end method CheckSize

} // end class UsingHugeIntegerService
```



Please enter two positive numbers up to 100 digits each:

```
3535353535353535353535353535353535353535353535353535353535353535
1212121212121212121212121212121212121212121212121212121212121212
4747474747474747474747474747474747474747474747474747474747474747
```

Fig. 21.13 Using the `HugeInteger` Web service. (Part 5 of 6.)



Please enter two positive numbers up to 100 digits each:

```
3535353535353535353535353535353535353535353535353535353535353535
1212121212121212121212121212121212121212121212121212121212121212
```

```
2323232323232323232323232323232323232323232323232323232323232323
```

Please enter two positive numbers up to 100 digits each:

```
3535353535353535353535353535353535353535353535353535353535353535
1212121212121212121212121212121212121212121212121212121212121212
```

```
3535353535353535353535353535353535353535353535353535353535353535 is larger than
1212121212121212121212121212121212121212121212121212121212121212
```

Fig. 21.13 Using the **`HugeInteger`** Web service. (Part 6 of 6.)

The user inputs two integers, each up to 100 digits long. The clicking of any button invokes a remote method to perform the appropriate calculation and return the result. The return value of each operation is displayed, and all leading zeroes are eliminated using **`string`** method **`TrimStart`**. Note that **`UsingHugeInteger`** does not have the capability to perform operations with 100-digit numbers. Instead, it creates **`string`** representations of these numbers and passes them as arguments to Web-service methods that handle such tasks for us.

21.5 Session Tracking in Web Services

In Chapter 20, ASP .NET, Web Forms and Web Controls, we described the importance of maintaining information about users to personalize their experiences. In the context of this discussion, we explored session tracking using cookies and sessions. In this section, we incorporate session tracking into a Web service. Sometimes, it makes sense for client applications to call several methods from the same Web service, and to call some methods several times. It would be beneficial for the Web service to maintain state information for the client. Using session tracking can be beneficial, because information that is stored as part of the session will not need to be passed back and forth between the Web service and the client. This will not only cause the client application to run faster, but it will require less effort on the part of the programmer (who likely will have to pass less information to a Web-service method).

Storing session information also can provide for a more intuitive Web service. In the following example, we create a Web service designed to assist with the computations involved in playing a game of Blackjack (Fig. 21.14). We then use this Web service to create a dealer for a game of Blackjack. This dealer handles the details for our deck of cards. The information is stored as part of the session, so that one set of cards does not get mixed up with another deck of cards being used by another client application. Our example uses casino Blackjack rules as follows:

> *Two cards each are dealt to each the dealer and the player. The player's cards are dealt face up. Only one of the dealer's cards is dealt face up. Then, the player can begin taking additional cards, one at a time. These cards are dealt face up, and the player decides when to stop taking cards. If the sum of the player's cards exceeds 21, the game is over, and the player loses. When the player is satisfied with the current set of cards, the player "stays" (i.e., stops taking cards), and the dealer's hidden card is revealed. If the dealer's total is less than 17, the dealer must take another card; otherwise, the dealer must stay. The dealer must continue to take cards until the sum of the dealer's cards is greater than or equal to 17. If the dealer exceeds 21, the player wins. Otherwise, the hand with the higher point total wins. If both sets of cards have the same point total, the game is a "push" (i.e., a tie), and no one wins. Finally, if a player's first two cards total 21, the player immediately wins. This type of win is known as a "Blackjack."*

The Web service that we create provides methods to deal a card and to count cards in a hand, determining a value for a specific hand. Each card is represented by a string in the form "**`face suit`**," where **`face`** is a digit that represents the face of the card, and **`suit`** is a digit that represents the suit of the card. After the Web service is created, we create a Windows application that uses these methods to implement a game of Blackjack.

```
// Fig. 21.14: BlackjackService.asmx.cs
// Blackjack Web Service which manipulates a deck of cards.

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace BlackjackWebService
{
   [ WebService(
      Namespace = "http://www.deitel.com/csphtp1/ch21/",
      Description = "A Web service that provides methods " +
      "to manipulate a deck of cards." ) ]
   public class BlackjackService : System.Web.Services.WebService
   {

      // Visual Studio .NET generated code

```

Fig. 21.14 **`Blackjack`** Web service. (Part 1 of 3.)

```
      // deal new card
      [ WebMethod( EnableSession = true,
         Description = "Deal a new card from the deck." ) ]
      public string DealCard()
      {
         string card = "2 2";

         // get client's deck
         ArrayList deck = ( ArrayList ) Session[ "deck" ];
         card = ( string ) deck[ 0 ];
         deck.RemoveAt( 0 );
         return card;

      } // end method DealCard

      [ WebMethod( EnableSession = true,
         Description = "Create and shuffle a deck of cards." ) ]
      public void Shuffle()
      {
         Random randomObject = new Random();

         ArrayList deck = new ArrayList();

         // generate all possible cards
         for ( int i = 2; i < 15; i++ ) {
            for ( int j = 0; j < 4; j++ ) {
               deck.Add( i + " " + j );
            }
         }

         // swap each card with another card randomly
         for ( int i = 0; i < deck.Count; i++ )
         {
            int newIndex = randomObject.Next( deck.Count );
            object temporary = deck[ i ];
            deck[ i ] = deck[ newIndex ];
            deck[ newIndex ] = temporary;
         }

         // add this deck to user's session state
         Session[ "deck" ] = deck;
      }

      // computes value of hand
      [ WebMethod ( Description = "Compute a " +
         "numerical value for the current hand." ) ]
      public int CountCards( string dealt )
      {
         // split string containing cards
         char[] tab = { '\t' };
         string[] cards = dealt.Split( tab );
         int total = 0, face, aceCount = 0;

```

Fig. 21.14 **`Blackjack`** Web service. (Part 2 of 3.)

```
76           foreach ( string drawn in cards )
77           {
78              // get face of card
79              face =
80                 Int32.Parse( drawn.Substring(
81                    0, drawn.IndexOf( " " ) ) );
82
83              switch ( face )
84              {
85                 // if ace, increment number of aces in hand
86                 case 14:
87                    aceCount++;
88                    break;
89
90                 // if Jack, Queen or King, add 10 to total
91                 case 11:  case 12:  case 13:
92                    total += 10;
93                    break;
94
95                 // otherwise, add value of face
96                 default:
97                    total += face;
98                    break;
99
100             } // end switch
101
102          } // end foreach
103
104          // if any aces, calculate optimum total
105          if ( aceCount > 0 )
106          {
107             // if it is possible to count one ace as 11, and rest
108             // 1 each, do so; otherwise, count all aces as 1 each
109             if ( total + 11 + aceCount - 1 <= 21 )
110                total += 11 + aceCount - 1;
111             else
112                total += aceCount;
113          }
114
115          return total;
116
117       }  // end method CountCards
118
119    } // end class BlackjackService
120
121 } // end namespace BlackjackWebService
```

Fig. 21.14 **`Blackjack`** Web service. (Part 3 of 3.)

Lines 24–36 define method **`DealCard`** as a **`WebMethod`**, with property ***`EnableSession`*** set to **`true`**. This property needs to be set to **`true`** to maintain session information. This simple step provides an important advantage to our Web service. The Web service now can use an **`HttpSessionState`** object (called **`Session`**) to maintain the deck of cards for each client application that wishes to use this Web service (line 31).

We can use **`Session`** to store objects for a specific client between method calls. We discussed session state in detail in Chapter 20, ASP .NET, Web Forms and Web Controls.

As we discuss shortly, method **`DealCard`** removes a card from the deck and returns it to the client. If we were not using a session variable, the deck of cards would need to be passed back and forth with each method call. Not only does the use of session state make the method easier to call (it now requires no arguments), but we avoid the overhead that would occur from sending this information back and forth, making our Web service faster.

In our current implementation, we simply have methods that use session variables. The Web service, however, still cannot determine which session variables belong to which user. This is an important point—if the Web service cannot uniquely identify a user, it has failed to perform session-tracking properly. If the same client called method **`DealCard`** twice, two different decks would be manipulated (as if two different users had called **`DealCard`**). To identify various users, the Web service creates a cookie for each user. Unfortunately, the Web service has no way of determining whether or not cookies are enabled on the client's machine. If the client application wishes to use this Web service, the client must accept this cookie in a ***`CookieContainer`*** object. We discuss this in more detail shortly, when we look into the client application that uses the **`Blackjack`** Web service.

Method **`DealCard`** (lines 24–36) obtains the current user's deck as an ***`ArrayList`*** from the Web service's **`Session`** object (line 31). You can think of an **`ArrayList`** as a dynamic array (i.e., its size can change at runtime). Class **`ArrayList`** is discussed in greater detail in Chapter 23, Data Structures and Collections. The class's method **`Add`** places an **`object`** in the **`ArrayList`**. Method **`DealCard`** then removes the top card from the deck (line 33) and returns the card's value as a **`string`** (line 34).

Method **`Shuffle`** (lines 38–64) generates an **`ArrayList`** representing a card deck, shuffles it and stores the shuffled cards in the client's **`Session`** object. Lines 47–51 include **`for`** loops to generate **`string`**s in the form "**`face suit`**" to represent each possible card in a deck. Lines 54–60 shuffle the re-created deck by swapping each card with another card in the deck. Line 63 adds the **`ArrayList`** to the **`Session`** object to maintain the deck between method calls.

Method **`CountCards`** (lines 67–117) counts the values of the cards in a hand by trying to attain the highest score possible without going over 21. Precautions need to be taken when calculating the value of the cards, because an ace can be counted as either 1 or 11, and all face cards count as 10.

The string **`dealt`** is tokenized into its individual cards by calling **`string`** method **`Split`** and passing it an array that contains the tab character. The **`foreach`** loop (line 76–102) counts the value of each card. Lines 79–81 retrieve the first integer—the face—and use that value as input to the **`switch`** statement in line 83. If the card is 1 (an ace), the program increments variable **`aceCount`**. Because an ace can have two values, additional logic is required to process aces. If the card is a 13, 12 or 11 (King, Queen or Jack), the program adds 10 to the total. If the card is anything else, the program increases the total by that value.

In lines 105–113, the aces are counted after all the other cards. If several aces are included in a hand, only one can be counted as 11 (e.g., if two were counted as 11 we would already have a hand value of 22, which is a losing hand). We then determine if we can count an ace as 11 without exceeding 21. If this is possible, line 110 adjusts the total accordingly. Otherwise, line 112 adjusts the total by counting each ace as 1 point.

`CountCards` attempts to maximize the value of the current cards without exceeding 21. Imagine, for example, that the dealer has a 7 and then receives an ace. The new total could be either 8 or 18. However, **`CountCards`** always tries the maximize the value of the cards without going over 21, so the new total is 18.

Now, we use the **`Blackjack`** Web service in a Windows application called **`Game`** (Fig. 21.15). This program uses an instance of **`BlackjackWebService`** to represent the dealer, calling its **`DealCard`** and **`CountCards`** methods. The Web service keeps track of both the player's and the dealer's cards (i.e., all the cards that have been dealt).

Each player has 11 **`PictureBox`**es—the maximum number of cards that can be dealt without exceeding 21. These **`PictureBox`**es are placed in an **`ArrayList`**, allowing us to index the **`ArrayList`** to determine which **`PictureBox`** displays the card image.

Previously we mentioned that the client must provide a way to accept any cookies created by the Web service to identify users. Line 64 in the constructor creates a new **`CookieContainer`** object for the **`CookieContainer`** property of **`dealer`**. Class **`CookieContainer`** (defined in namespace **`System.Net`**) acts as a storage space for an object of the **`HttpCookie`** class. Creating the **`CookieContainer`** allows the Web service to maintain session state for the current client. This **`CookieContainer`** stores a **`Cookie`** with a unique identifier that the server can use to recognize the client when the client makes future requests. By default, the **`CookieContainer`** is **`null`**, and a new **`Session`** object is created by the Web Service for each client.

```
1  // Fig. 21.15: Blackjack.cs
2  // Blackjack game that uses the Blackjack Web service.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9  using System.Data;
10 using System.Net;
11
12 // game that uses Blackjack Web Service
13 public class Blackjack : System.Windows.Forms.Form
14 {
15    private System.Windows.Forms.PictureBox pictureBox1;
16    private System.Windows.Forms.PictureBox pictureBox2;
17    private System.Windows.Forms.PictureBox pictureBox3;
18    private System.Windows.Forms.PictureBox pictureBox4;
19    private System.Windows.Forms.PictureBox pictureBox5;
20    private System.Windows.Forms.PictureBox pictureBox6;
21    private System.Windows.Forms.PictureBox pictureBox7;
22    private System.Windows.Forms.PictureBox pictureBox8;
23    private System.Windows.Forms.PictureBox pictureBox9;
24    private System.Windows.Forms.PictureBox pictureBox10;
25    private System.Windows.Forms.PictureBox pictureBox11;
26    private System.Windows.Forms.PictureBox pictureBox12;
27    private System.Windows.Forms.PictureBox pictureBox13;
28    private System.Windows.Forms.PictureBox pictureBox14;
```

Fig. 21.15 Blackjack game that uses **`Blackjack`** Web service. (Part 1 of 8.)

```
      private System.Windows.Forms.PictureBox pictureBox15;
      private System.Windows.Forms.PictureBox pictureBox16;
      private System.Windows.Forms.PictureBox pictureBox17;
      private System.Windows.Forms.PictureBox pictureBox18;
      private System.Windows.Forms.PictureBox pictureBox19;
      private System.Windows.Forms.PictureBox pictureBox20;
      private System.Windows.Forms.PictureBox pictureBox21;
      private System.Windows.Forms.PictureBox pictureBox22;

      private System.Windows.Forms.Button dealButton;
      private System.Windows.Forms.Button hitButton;
      private System.Windows.Forms.Button stayButton;

      private System.ComponentModel.Container components = null;

      private localhost.BlackjackService dealer;
      private string dealersCards, playersCards;
      private ArrayList cardBoxes;
      private int playerCard, dealerCard;

      // labels displaying game status, dealer and player
      private System.Windows.Forms.Label dealerLabel;
      private System.Windows.Forms.Label playerLabel;
      private System.Windows.Forms.Label statusLabel;

      public enum GameStatus :
         int { PUSH, LOSE, WIN, BLACKJACK };

      public Blackjack()
      {
         InitializeComponent();

         dealer = new localhost.BlackjackService();

         // allow session state
         dealer.CookieContainer = new CookieContainer();

         cardBoxes = new ArrayList();

         // put PictureBoxes into cardBoxes
         cardBoxes.Add( pictureBox1 );
         cardBoxes.Add( pictureBox2 );
         cardBoxes.Add( pictureBox3 );
         cardBoxes.Add( pictureBox4 );
         cardBoxes.Add( pictureBox5 );
         cardBoxes.Add( pictureBox6 );
         cardBoxes.Add( pictureBox7 );
         cardBoxes.Add( pictureBox8 );
         cardBoxes.Add( pictureBox9 );
         cardBoxes.Add( pictureBox10 );
         cardBoxes.Add( pictureBox11 );
         cardBoxes.Add( pictureBox12 );
         cardBoxes.Add( pictureBox13 );
```

Fig. 21.15 Blackjack game that uses `Blackjack` Web service. (Part 2 of 8.)

```
         cardBoxes.Add( pictureBox14 );
         cardBoxes.Add( pictureBox15 );
         cardBoxes.Add( pictureBox16 );
         cardBoxes.Add( pictureBox17 );
         cardBoxes.Add( pictureBox18 );
         cardBoxes.Add( pictureBox19 );
         cardBoxes.Add( pictureBox20 );
         cardBoxes.Add( pictureBox21 );
         cardBoxes.Add( pictureBox22 );

      } // end method Blackjack

      // Visual Studio .NET generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new Blackjack() );

      } // end Main

      // deals cards to dealer while dealer's total is
      // less than 17, then computes value of each hand
      // and determines winner
      protected void stayButton_Click(
         object sender, System.EventArgs e )
      {
         stayButton.Enabled = false;
         hitButton.Enabled = false;
         dealButton.Enabled = true;
         DealerPlay();
      }

      // process dealers turn
      private void DealerPlay()
      {
         // while value of dealer's hand is below 17,
         // dealer must take cards
         while ( dealer.CountCards( dealersCards ) < 17 )
         {
            dealersCards += "\t" + dealer.DealCard();
            DisplayCard( dealerCard, "" );
            dealerCard++;
            MessageBox.Show( "Dealer takes a card" );
         }

         int dealersTotal = dealer.CountCards( dealersCards );
         int playersTotal = dealer.CountCards( playersCards );

         // if dealer busted, player wins
         if ( dealersTotal > 21 )
         {
            GameOver( GameStatus.WIN );
```

Fig. 21.15 Blackjack game that uses `Blackjack` Web service. (Part 3 of 8.)

```
            return;
         }

         // if dealer and player have not exceeded 21,
         // higher score wins; equal scores is a push.
         if ( dealersTotal > playersTotal )
            GameOver( GameStatus.LOSE );
         else if ( playersTotal > dealersTotal )
            GameOver( GameStatus.WIN );
         else
            GameOver( GameStatus.PUSH );

      } // end method DealerPlay

      // deal another card to player
      protected void hitButton_Click(
         object sender, System.EventArgs e )
      {
         // get player another card
         string card = dealer.DealCard();
         playersCards += "\t" + card;
         DisplayCard( playerCard, card );
         playerCard++;

         int total = dealer.CountCards( playersCards );

         // if player exceeds 21, house wins
         if ( total > 21 )
            GameOver( GameStatus.LOSE );

         // if player has 21, they cannot take more cards
         // the dealer plays
         if ( total == 21 )
         {
            hitButton.Enabled = false;
            DealerPlay();
         }

      } // end method hitButton_Click

      // deal two cards each to dealer and player
      protected void dealButton_Click(
         object sender, System.EventArgs e )
      {
         string card;

         // clear card images
         foreach ( PictureBox cardImage in cardBoxes )
            cardImage.Image = null;

         // clear status from previous game
         statusLabel.Text = "";

```

Fig. 21.15 Blackjack game that uses **Blackjack** Web service. (Part 4 of 8.)

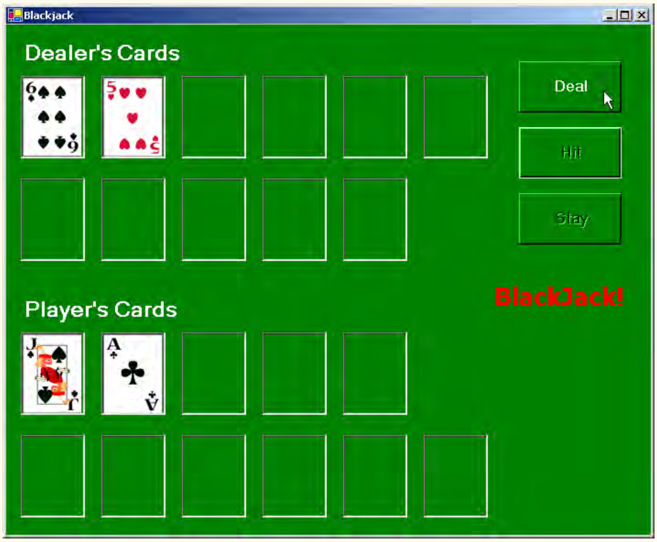
```
         // shuffle cards
         dealer.Shuffle();

         // deal two cards to player
         playersCards = dealer.DealCard();
         DisplayCard( 11, playersCards );
         card = dealer.DealCard();
         DisplayCard( 12, card );
         playersCards += "\t" + card;

         // deal two cards to dealer, only display face
         // of first card
         dealersCards = dealer.DealCard() ;
         DisplayCard( 0, dealersCards );
         card = dealer.DealCard();
         DisplayCard( 1, "" );
         dealersCards += "\t" + card;

         stayButton.Enabled = true;
         hitButton.Enabled = true;
         dealButton.Enabled = false;

         int dealersTotal = dealer.CountCards( dealersCards );
         int playersTotal = dealer.CountCards( playersCards );

         // if hands equal 21, it is a push
         if ( dealersTotal == playersTotal &&
            dealersTotal == 21 )
            GameOver( GameStatus.PUSH );

         // if player has 21 player wins with blackjack
         else if ( playersTotal == 21 )
            GameOver( GameStatus.BLACKJACK );

         // if dealer has 21, dealer wins
         else if ( dealersTotal == 21 )
            GameOver( GameStatus.LOSE );

         dealerCard = 2;
         playerCard = 13;

      } // end method dealButton_Click

      // displays card represented by cardValue in
      // PictureBox with number card
      public void DisplayCard( int card, string cardValue )
      {
         // retrieve appropriate PictureBox from ArrayList
         PictureBox displayBox = ( PictureBox ) cardBoxes[ card ];

```

Fig. 21.15 Blackjack game that uses **`Blackjack`** Web service. (Part 5 of 8.)

```
      // if string representing card is empty,
      // set displayBox to display back of card
      if ( cardValue == "" )
      {
         displayBox.Image =
            Image.FromFile( "blackjack_images\\cardback.png" );
         return;
      }

      // retrieve face value of card from cardValue
      int faceNumber = Int32.Parse( cardValue.Substring( 0,
         cardValue.IndexOf( " " ) ) );

      string face = faceNumber.ToString();

      // retrieve the suit of the card from cardValue
      string suit = cardValue.Substring(
         cardValue.IndexOf( " " ) + 1 );

      char suitLetter;

      // determine if suit is other than clubs
      switch ( Convert.ToInt32( suit ) )
      {
         // suit is clubs
         case 0:
            suitLetter = 'c';
            break;

         // suit is diamonds
         case 1:
            suitLetter = 'd';
            break;

         // suit is hearts
         case 2:
            suitLetter = 'h';
            break;

         // else suit is spades
         default:
            suitLetter = 's';
            break;
      }

      // set displayBox to display appropriate image
      displayBox.Image = Image.FromFile(
         "blackjack_images\\" + face + suitLetter + ".png"  );

   } // end method DisplayCard

   // displays all player cards and shows
   // appropriate game status message
```

Fig. 21.15 Blackjack game that uses **`Blackjack`** Web service. (Part 6 of 8.)

```
   public void GameOver( GameStatus winner )
   {
      char[] tab = { '\t' };
      string[] cards = dealersCards.Split( tab );

      for ( int i = 0; i < cards.Length; i++ )
         DisplayCard(  i, cards[ i ] );

      // push
      if ( winner == GameStatus.PUSH )
         statusLabel.Text = "It's a tie!";

      // player loses
      else if ( winner == GameStatus.LOSE )
         statusLabel.Text = "You Lose Try Again!";

      // player wins
      else if ( winner == GameStatus.WIN )
         statusLabel.Text = "You Win!";

      // player has won with blackjack
      else
         statusLabel.Text = "BlackJack!";

      stayButton.Enabled = false;
      hitButton.Enabled = false;
      dealButton.Enabled = true;

   } // end method GameOver

} // end class Blackjack
```



BlackJack!

Fig. 21.15 Blackjack game that uses `Blackjack` Web service. (Part 7 of 8.)

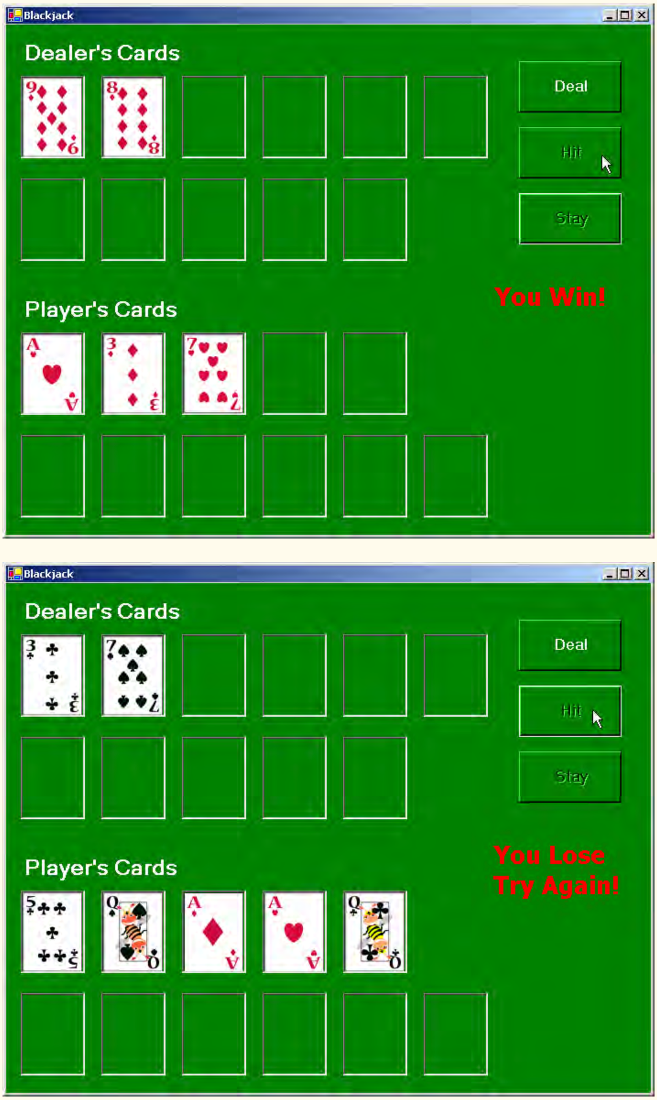

Blackjack
Dealer's Cards
Deal
Hit
Stay
You Win!
Player's Cards
Blackjack
Dealer's Cards
Deal
Hit
Stay
You Lose
Try Again!
Player's Cards

Fig. 21.15 Blackjack game that uses `Blackjack` Web service. (Part 8 of 8.)

Method **`GameOver`** (lines 291–319) displays all the dealer's cards (many of which are turned face down during the game) and shows the appropriate message in the status **`PictureBox`**. Method **`GameOver`** receives as an argument a member of the **`GameStatus`** enumeration (defined in lines 54–55). The enumeration represents whether the player tied, lost or won the game; its four members are: **`PUSH`**, **`LOSE`**, **`WIN`** and **`BLACKJACK`**.

When the player clicks the **Deal** button (in the event handler on lines 176–229), all the **`PictureBox`**es are cleared, the deck is shuffled and the player and dealer receive two cards each. If both obtain scores of 21, method **`GameOver`** is called and is passed **`GameStatus.PUSH`**. If the player has 21, **`GameOver`** is called and is passed **`GameStatus.BLACKJACK`**. Finally, if only the dealer has 21, method **`GameOver`** is called and is passed **`GameStatus.LOSE`**.

If **`GameOver`** is not called, the player can take additional cards by clicking the **Hit** button (in the event handler on lines 150–173). Each time a player clicks **Hit**, the player is dealt one card, which is displayed in the GUI. If the player exceeds 21, the game is over, and the player loses. If the player has exactly 21, the player is not allowed to take any more cards.

Players can click the **Stay** button to indicate that they do not want to risk being dealt another card. In the event handler for this event (lines 106–113), all three buttons are disabled, and method **`DealerPlay`** is called. This method (lines 116–147) causes the dealer to keep taking cards until the dealer's hand is worth 17 or more. If the dealer's hand exceeds 21, the player wins; otherwise, the values of the hands are compared, and **`GameOver`** is called with the appropriate argument.

Method **`DisplayCard`** (lines 233–287) retrieves the appropriate card image. It takes as arguments an integer representing the index of the **`PictureBox`** in the **`ArrayList`** that must have its image set and a **`string`** representing the card. An empty **`string`** indicates that we wish to display the back of a card; otherwise, the program extracts the face and suit from the **`string`** and uses this information to find the correct image. The **`switch`** statement (lines 260–281) converts the number representing the suit into an integer and assigns the appropriate character to **`suitLetter`** (**`c`** for Clubs, **`d`** for Diamonds, **`h`** for Hearts and **`s`** for Spades). The character **`suitLetter`** completes the image's file name.

21.6 Using Web Forms and Web Services

In the previous examples, we have accessed Web services from Windows applications. However, we can just as easily use them in Web applications. Because Web-based business is becoming more and more prevalent, it often is more practical for programmers to design Web services as part of Web applications. Figure 21.16 presents an airline-reservation Web service that receives information regarding the type of seat the customer wishes to reserve and then makes a reservation if such a seat is available.

The airline-reservation Web service has a single **`WebMethod`**—**`Reserve`** (lines 36–85)—which searches its seat database to locate a seat matching a user's request. If it finds an appropriate seat, **`Reserve`** updates the database, makes the reservation and returns **`true`**; otherwise, no reservation is made, and the method returns **`false`**.

`Reserve` takes two arguments: A **`string`** representing the desired type of seat (the choices are window, middle or aisle) and a **`string`** representing the desired class type (the choices are economy or first class). Our database contains four columns: The seat number, the seat type, the class type and a column containing either 0 or 1 to indicate whether the seat is taken. Lines 48–51 define an SQL command that retrieves the number of available seats matching the requested seat and class types. The statement in lines 52–53 executes the query. If the result of the query is not empty, the application reserves the first seat number that the query returns. The database is updated with an **`UPDATE`** command, and **`Reserve`** returns **`true`**, indicating that the reservation was successful. If the result of the **`SELECT`** query is not successful, **`Reserve`** returns **`false`**, indicating that no available seats matched the request.

```
// Fig. 21.16: Reservation.asmx.cs
// Airline reservation Web Service.

using System;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Data.OleDb;

namespace AirlineReservation
{
   // performs reservation of a seat
   [ WebService( Namespace = "http://www.deitel.com/csphtp1/ch21/",
      Description = "Service that enables a user to " +
      "reserve a seat on a plane." ) ]
   public class Reservation : System.Web.Services.WebService
   {
      private System.Data.OleDb.OleDbCommand
         oleDbSelectCommand1;
      private System.Data.OleDb.OleDbCommand
         oleDbInsertCommand1;
      private System.Data.OleDb.OleDbCommand
         oleDbUpdateCommand1;
      private System.Data.OleDb.OleDbCommand
         oleDbDeleteCommand1;
      private System.Data.OleDb.OleDbConnection
         oleDbConnection1;
      private System.Data.OleDb.OleDbDataAdapter
         oleDbDataAdapter1;

      // Visual Studio .NET generated code

      // checks database to determine whether
      // matching seat is available
      [ WebMethod ( Description = "Method to reserve seat." ) ]
      public bool Reserve( string seatType, string classType )
      {
         OleDbDataReader dataReader;

         // try database connection
         try
         {
            // open database connection
            oleDbConnection1.Open();

            // set and execute SQL query
            oleDbDataAdapter1.SelectCommand.CommandText =
               "SELECT Number FROM Seats WHERE Type = '" +
               seatType + "' AND Class = '" + classType +
               "' AND Taken = '0'" ;
            dataReader =
               oleDbDataAdapter1.SelectCommand.ExecuteReader();
```

Fig. 21.16 Airline reservation Web service. (Part 1 of 2.)

```

            // if there were results, seat is available
            if ( dataReader.Read() )
            {
               string seatNumber = dataReader.GetString( 0 );

               dataReader.Close();

               // update first available seat to be taken
               oleDbDataAdapter1.UpdateCommand.CommandText =
                  "Update Seats Set Taken = '1' WHERE Number = '"
                  + seatNumber + "'";
               oleDbDataAdapter1.UpdateCommand.ExecuteNonQuery();

               return true;

            } // end if
            dataReader.Close();
         }
         catch ( OleDbException ) // if connection problem
         {
            return false;
         }
         finally
         {
            oleDbConnection1.Close();
         }

         // no seat was reserved
         return false;

      } // end method Reserve

   } // end class Reservation

} // end namespace AirlineReservation
```

Fig. 21.16 Airline reservation Web service. (Part 2 of 2.)

Earlier in the chapter, we displayed a Web service in design view (Fig. 21.7), and we explained that this design view allows the programmer to add components to a Web service. In our airline-reservation Web service (Fig. 21.16), we used various data components. Figure 21.18 shows these components in design view. Notice that it is easier to drop these components into our Web service using the **Toolbox** than to type the equivalent code.

Figure 21.18 presents the ASPX listing for the Web Form through which users can select seat types. This page allows a user to reserve a seat on the basis of its class and location in a row of seats. The page then uses the airline-reservation Web service to carry out the user's request. If the database request is not successful, the user is instructed to modify the request and try again.

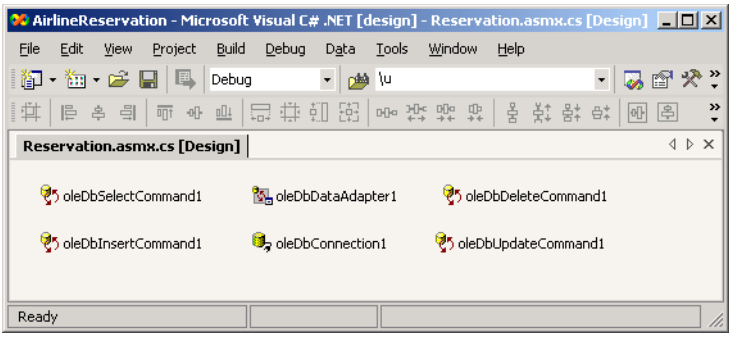

AirlineReservation - Microsoft Visual C# .NET [design] - Reservation.asmx.cs [Design]
File
Edit
View
Project
Build
Debug
Data
Tools
Window
Help
Debug
\u
Reservation.asmx.cs [Design]
oleDbSelectCommand1
oleDbDataAdapter1
oleDbDeleteCommand1
oleDbInsertCommand1
oleDbConnection1
oleDbUpdateCommand1
Ready

Fig. 21.17 Airline Web Service in design view.

```
<%-- Fig. 21.18: TicketReservation.aspx                   --%>
<%-- A Web Form to allow users to select the kind of seat --%>
<%-- they wish to reserve.                                --%>

<%@ Page language="c#" Codebehind="TicketReservation.aspx.cs"
    AutoEventWireup="false"
    Inherits="MakeReservation.TicketReservation" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
   <HEAD>
      <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
      <meta name="CODE_LANGUAGE" Content="C#">
      <meta name="vs_defaultClientScript"
            content="JavaScript (ECMAScript)">
      <meta name="vs_targetSchema"
          content="http://schemas.microsoft.com/intellisense/ie5">
   </HEAD>
   <body MS_POSITIONING="GridLayout">

      <form id="MakeReservation" method="post" runat="server">

         <asp:DropDownList id="seatList" style="Z-INDEX: 101;
            LEFT: 16px; POSITION: absolute; TOP: 43px"
            runat="server" Width="105px" Height="22px">

            <asp:ListItem Value="Aisle">Aisle</asp:ListItem>
            <asp:ListItem Value="Middle">Middle</asp:ListItem>
            <asp:ListItem Value="Window">Window</asp:ListItem>

         </asp:DropDownList>

         <asp:DropDownList id="classList" style="Z-INDEX: 102;
            LEFT: 145px; POSITION: absolute; TOP: 43px"
            runat="server" Width="98px" Height="22px">

```

Fig. 21.18 ASPX file that takes reservation information. (Part 1 of 2.)

```
            <asp:ListItem Value="Economy">Economy</asp:ListItem>
            <asp:ListItem Value="First">First</asp:ListItem>

         </asp:DropDownList>

         <asp:Button id="reserveButton" style="Z-INDEX: 103;
          LEFT: 21px; POSITION: absolute; TOP: 83px" runat="server"
            Text="Reserve">
         </asp:Button>

         <asp:Label id="Label1" style="Z-INDEX: 104;
            LEFT: 17px; POSITION: absolute; TOP: 13px"
            runat="server">Please select the type of seat and
            class you wish to reserve:
         </asp:Label>

      </form>
   </body>
</HTML>
```

Fig. 21.18 ASPX file that takes reservation information. (Part 2 of 2.)

The page in Fig. 21.17 defines two **`DropDownList`** objects and a **`Button`**. One **`DropDownList`** displays all the seat types from which users can select. The second lists choices for the class type. Users click the **`Button`**, named **`reserveButton`**, to submit requests after making selections from the **`DropDownList`**s. The code-behind file (Fig. 21.19) attaches an event handler for this button.

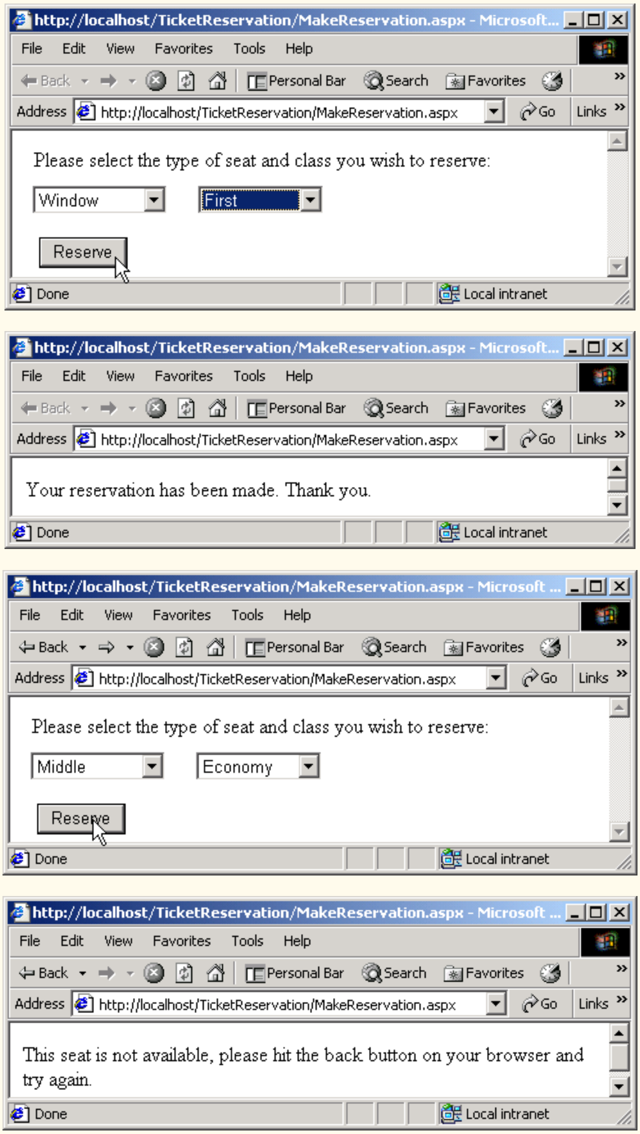
Lines 30–31 create a **`Reservation`** object. When the user clicks **Reserve**, the **`reserveButton_Click`** event handler executes, and the page reloads. The event handler (lines 48–63) calls the Web service's **`Reserve`** method and passes it the selected seat and class types as arguments. If **`Reserve`** returns **`true`**, the application displays a message thanking the user for making a reservation; otherwise, the user is notified that the type of seat requested is not available, and the user is instructed to try again.

```
1  // Fig. 21.19: TicketReservation.aspx.cs
2  // Making a Reservation using a Web Service.
3
4  using System;
5  using System.Collections;
6  using System.ComponentModel;
7  using System.Data;
8  using System.Drawing;
9  using System.Web;
10 using System.Web.SessionState;
11 using System.Web.UI;
12 using System.Web.UI.WebControls;
13 using System.Web.UI.HtmlControls;
14
```

Fig. 21.19 Code-behind file for the reservation page. (Part 1 of 3.)

```
namespace MakeReservation
{
   // allows visitors to select seat type to reserve, and
   // then make reservation
   public class TicketReservation : System.Web.UI.Page
   {
      protected System.Web.UI.WebControls.DropDownList
         seatList;
      protected System.Web.UI.WebControls.DropDownList
         classList;

      protected System.Web.UI.WebControls.Button
         reserveButton;
      protected System.Web.UI.WebControls.Label Label1;

      private localhost.Reservation agent =
         new localhost.Reservation();

      private void Page_Load(
         object sender, System.EventArgs e )
      {
         if ( IsPostBack )
         {
            seatList.Visible = false;
            classList.Visible = false;
            reserveButton.Visible = false;
            Label1.Visible = false;
         }
      }

      // Visual Studio .NET generated code

      // calls Web Service to try to reserve specified seat
      public void reserveButton_Click (
         object sender, System.EventArgs e )
      {
         // if Web-service method returned true, signal success
         if ( agent.Reserve( seatList.SelectedItem.Text,
            classList.SelectedItem.Text ) )
            Response.Write( "Your reservation has been made."
               + "  Thank you." );

         // Web-service method returned false, so signal failure
         else
            Response.Write( "This seat is not available, " +
               "please hit the back button on your browser " +
               "and try again." );

      } // end method reserveButton_Click

   } // end class TicketReservation

} // end namespace MakeReservation
```

Fig. 21.19 Code-behind file for the reservation page. (Part 2 of 3.)


http://localhost/TicketReservation/MakeReservation.aspx - Microsoft...
File Edit View Favorites Tools Help
Back
Personal Bar
Search
Favorites
Address
Go
Links
Reserve
Done
Local intranet

Address: http://localhost/TicketReservation/MakeReservation.aspx

Please select the type of seat and class you wish to reserve:

Window First

Your reservation has been made. Thank you.

Please select the type of seat and class you wish to reserve:

Middle Economy

This seat is not available, please hit the back button on your browser and try again.

Fig. 21.19 Code-behind file for the reservation page. (Part 3 of 3.)

21.7 Case Study: Temperature Information Application

This case study discusses both a Web service that presents weather forecasts for various cities around the United States and a windows application that employs the Web service. The Web service uses networking capabilities to display the forecasts; it parses a Web page containing the required information and then extracts weather forecast data.

First, we present Web service **`TemperatureServer`** in Fig. 21.20. This Web service reads a Web page and collects information about the temperature and weather conditions in an assortment of American cities. [*Note*: At the time of publication, this program runs in the manner that we describe. However, if changes are made to the Web page from which the program retrieves data, the program might work differently or not at all. Please check our Web site at **`www.deitel.com`** for updates.]

```
// Fig. 21.20: TemperatureServer.asmx.cs
// TemperatureServer Web Service that extracts weather
// information from a Web page.

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.IO;
using System.Net;

namespace TemperatureWebService
{
   [ WebService( Namespace = "http://www.deitel.com/csphtp1/ch21/",
      Description = "A Web service that provides information " +
      "from the National Weather Service." ) ]
   public class TemperatureServer :
      System.Web.Services.WebService
   {
      // Visual Studio .NET generated code

      [ WebMethod( EnableSession = true, Description =
         "Method to read information from the weather service." ) ]
      public void UpdateWeatherConditions()
      {
         // create WebClient to get access to Web page
         WebClient myClient = new WebClient();
         ArrayList cityList = new ArrayList();

         // get StreamReader for response so we can read page
         StreamReader input = new StreamReader(
            myClient.OpenRead(
            "http://iwin.nws.noaa.gov/iwin/us/" +
            "traveler.html" ) );

         string separator = "TAV12";

         // locate first horizontal line on Web page
         while ( !input.ReadLine().StartsWith(
            separator ) ) ; // do nothing

```

Fig. 21.20 **`TemperatureServer`** Web service. (Part 1 of 3.)

```
         // day format and night format
         string dayFormat =
            "CITY            WEA     HI/LO   WEA     " +
            "HI/LO";
         string nightFormat =
            "CITY            WEA     LO/HI   WEA     " +
            "LO/HI";
         string inputLine = "";

         // locate header that begins weather information
         do
         {
            inputLine = input.ReadLine();
         } while ( !inputLine.Equals( dayFormat ) &&
            !inputLine.Equals( nightFormat ) );

         // get first city's data
         inputLine = input.ReadLine();

         while ( inputLine.Length > 28 )
         {
            // create CityWeather object for city
            CityWeather weather = new CityWeather(
               inputLine.Substring( 0, 16 ),
               inputLine.Substring( 16, 7 ),
               inputLine.Substring( 23, 7 ) );

            // add to List
            cityList.Add( weather );

            // get next city's data
            inputLine = input.ReadLine();
         }

         // close connection to NWS server
         input.Close();

         // add city list to user session
         Session.Add( "cityList", cityList );

      } // end UpdateWeatherConditions

      // gets all city names
      [ WebMethod( EnableSession = true, Description =
         "Method to retrieve a list of cities." ) ]
      public string[] Cities()
      {
         ArrayList cityList = ( ArrayList ) Session[ "cityList" ];
         string[] cities= new string[ cityList.Count ];

         // retrieve names for cities
         for ( int i = 0; i < cityList.Count; i++ )
         {
```

Fig. 21.20 **`TemperatureServer`** Web service. (Part 2 of 3.)

```
            CityWeather weather = ( CityWeather ) cityList[ i ];

            cities[ i ] = weather.CityName;
         }

         return cities;

      } // end method Cities

      // gets all city descriptions
      [ WebMethod( EnableSession = true, Description = "Method" +
         " to retrieve weather descriptions for a " +
         "list of cities." )]
      public string[] Descriptions()
      {
         ArrayList cityList = ( ArrayList ) Session[ "cityList" ];
         string[] descriptions= new string[ cityList.Count ];

         // retrieve weather descriptions for all cities
         for ( int i = 0; i < cityList.Count; i++ )
         {
            CityWeather weather = ( CityWeather )cityList[ i ];

            descriptions[ i ] = weather.Description;
         }

         return descriptions;

      } // end method Descriptions

      // obtains each city temperature
      [ WebMethod( EnableSession = true, Description = "Method " +
         "to retrieve the temperature for a list of cities." ) ]
      public string[] Temperatures()
      {
         ArrayList cityList = ( ArrayList ) Session[ "cityList" ];
         string[] temperatures= new string[ cityList.Count ];

         // retrieve temperatures for all cities
         for ( int i = 0; i < cityList.Count; i++ )
         {
            CityWeather weather = ( CityWeather )cityList[ i ];
            temperatures[ i ] = weather.Temperature;
         }

         return temperatures;

      } // end method Temperatures

   } // end class TemperatureServer

} // end namespace TemperatureWebService
```

Fig. 21.20 **`TemperatureServer`** Web service. (Part 3 of 3.)

Method **UpdateWeatherConditions**, which gathers weather data from a Web page, is the first **WebMethod** that a client must call from the Web service. The service also provides **WebMethod**s **Cities**, **Descriptions** and **Temperatures**, which return different kinds of forecast-related information.

When **UpdateWeatherConditions** (lines 25–85) is invoked, the method connects to a Web site containing the traveler's forecasts from the National Weather Service (NWS). Line 30 creates a ***WebClient*** object, which we use because the **WebClient** class is designed for interaction with a source specified by a URL. In this case, the URL for the NWS page is **http://iwin.nws.noaa.gov/iwin/us/traveler.html**. Lines 34–37 call **WebClient** method ***OpenRead***; the method retrieves a **Stream** from the URL containing the weather information and then uses this **Stream** to create a **StreamReader** object. Using a **StreamReader** object, the program can read the Web page's HTML markup line by line.

The section of the Web page in which we are interested starts with the **string** "**TAV12**." Therefore, lines 42–43 read the HTML markup one line at a time until this **string** is encountered. Once the string "**TAV12**" is reached, the **do**/**while** structure (lines 55–59) continues to read the page one line at a time until it finds the header line (i.e., the line at the beginning of the forecast table). This line starts with either **dayFormat**, indicating day format, or **nightFormat**, indicating night format. Because the line could be in either format, the structure checks for both. Line 62 reads the next line from the page, which is the first line containing temperature information.

The **while** structure (lines 64–77) creates a new **CityWeather** object to represent the current city. It parses the **string** containing the current weather data, separating the city name, the weather condition and the temperature. The **CityWeather** object is added to **cityList** (an **ArrayList** that contains a list of the cities, their descriptions and their current temperatures); then, the next line from the page is read and stored in **inputLine** for the next iteration. This process continues until the length of the **string** read from the Web page is less than or equal to **28**. This signals the end of the temperature section. Line 83 adds the **ArrayList cityList** to the **Session** object so that the values are maintained between method calls.

Method **Cities** (lines 88–105) creates an array of **string**s that can contain as many **string** elements as there are elements in **cityList**. Line 92 obtains the list of cities from the **Session** object. Lines 96–101 iterate through each **CityWeather** object in **cityList** and insert the city name into the array, which is returned in line 103. Methods **Descriptions** (lines 108–126) and **Temperatures** (lines 129–145) behave similarly, except that they return weather descriptions and temperatures, respectively.

Figure 21.21 contains the code listing for the **CityWeather** class. The constructor takes three arguments: The city's name, the weather description and the current temperature. The class provides the read-only properties **CityName**, **Temperature** and **Description** so that these values can be retrieved by the Web service.

```
// Fig. 21.21: CityWeather.cs
// Class representing the weather information for one city.

using System;
```

Fig. 21.21 Class that stores weather information about a city. (Part 1 of 2.)

```

namespace TemperatureWebService
{
   public class CityWeather
   {
      private string cityName;
      private string temperature;
      private string description;

      public CityWeather(
         string city, string information, string degrees )
      {
         cityName = city;
         description = information;
         temperature = degrees;
      }

      // city name
      public string CityName
      {
         get
         {
            return cityName;
         }
      }

      // city temperature
      public string Temperature
      {
         get
         {
            return temperature;
         }
      }

      // forecast description
      public string Description
      {
         get
         {
            return description;
         }
      }

   } // end class CityWeather
} // end namespace TemperatureWebService
```

Fig. 21.21 Class that stores weather information about a city. (Part 2 of 2.)

The Windows application in Fig. 21.22 uses the **`TemperatureServer`** Web service to display weather information in a user-friendly format.

`TemperatureClient` (Fig. 21.22) is a Windows application that uses the **`TemperatureServer`** Web service to display weather information in a graphical and easy-to-read manner. The application consists of 36 **Label**s, which are placed in two columns. Each **Label** displays the weather information for a different city.

```
// Fig. 21.22: Client.cs
// Class that displays weather information that it receives
// from a Web service.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Net;

namespace TemperatureClient
{
   public class Client : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Label label1;
      private System.Windows.Forms.Label label2;
      private System.Windows.Forms.Label label3;
      private System.Windows.Forms.Label label4;
      private System.Windows.Forms.Label label5;
      private System.Windows.Forms.Label label6;
      private System.Windows.Forms.Label label7;
      private System.Windows.Forms.Label label8;
      private System.Windows.Forms.Label label9;
      private System.Windows.Forms.Label label10;
      private System.Windows.Forms.Label label11;
      private System.Windows.Forms.Label label12;
      private System.Windows.Forms.Label label13;
      private System.Windows.Forms.Label label14;
      private System.Windows.Forms.Label label15;
      private System.Windows.Forms.Label label16;
      private System.Windows.Forms.Label label17;
      private System.Windows.Forms.Label label18;
      private System.Windows.Forms.Label label19;
      private System.Windows.Forms.Label label20;
      private System.Windows.Forms.Label label21;
      private System.Windows.Forms.Label label22;
      private System.Windows.Forms.Label label23;
      private System.Windows.Forms.Label label24;
      private System.Windows.Forms.Label label25;
      private System.Windows.Forms.Label label26;
      private System.Windows.Forms.Label label27;
      private System.Windows.Forms.Label label28;
      private System.Windows.Forms.Label label29;
      private System.Windows.Forms.Label label30;
      private System.Windows.Forms.Label label31;
      private System.Windows.Forms.Label label32;
      private System.Windows.Forms.Label label33;
      private System.Windows.Forms.Label label34;
```

Fig. 21.22 Receiving temperature and weather data from a Web service. (Part 1 of 4.)

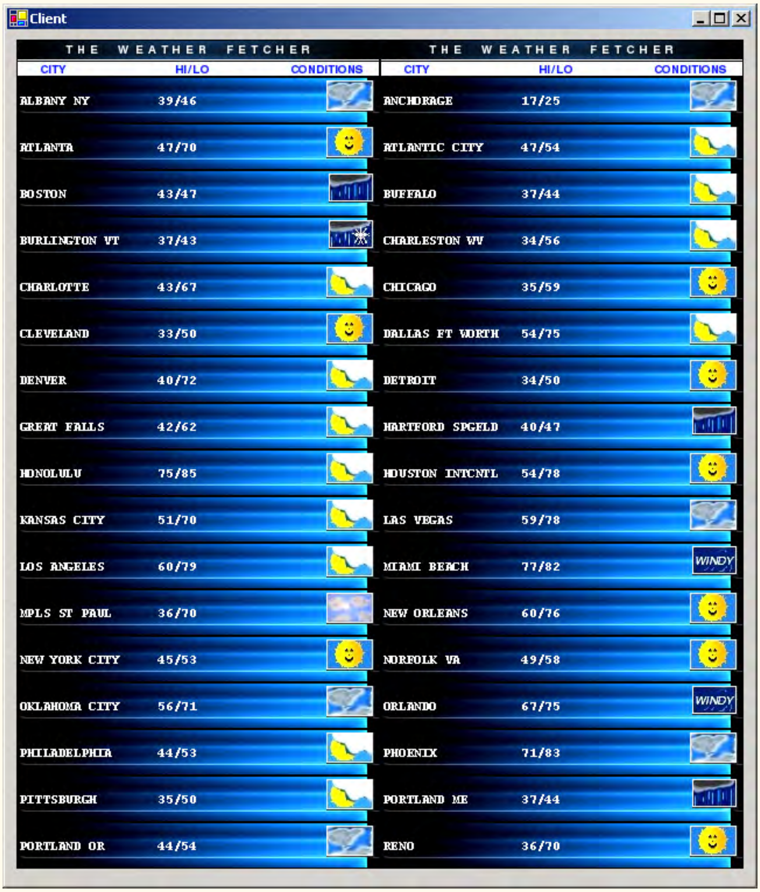
```
      private System.Windows.Forms.Label label36;
      private System.Windows.Forms.Label label35;

      private System.ComponentModel.Container components =
         null;

      public Client()
      {
         InitializeComponent();

         localhost.TemperatureServer client =
            new localhost.TemperatureServer();
         client.CookieContainer = new CookieContainer();
         client.UpdateWeatherConditions();

         string[] cities = client.Cities();
         string[] descriptions = client.Descriptions();
         string[] temperatures = client.Temperatures();

         label35.BackgroundImage = new Bitmap(
            "images/header.png" );
         label36.BackgroundImage = new Bitmap(
            "images/header.png" );

         // create Hashtable and populate it with every label
         Hashtable cityLabels = new Hashtable();
         cityLabels.Add( 1, label1 );
         cityLabels.Add( 2, label2 );
         cityLabels.Add( 3, label3 );
         cityLabels.Add( 4, label4 );
         cityLabels.Add( 5, label5 );
         cityLabels.Add( 6, label6 );
         cityLabels.Add( 7, label7 );
         cityLabels.Add( 8, label8 );
         cityLabels.Add( 9, label9 );
         cityLabels.Add( 10, label10 );
         cityLabels.Add( 11, label11 );
         cityLabels.Add( 12, label12 );
         cityLabels.Add( 13, label13 );
         cityLabels.Add( 14, label14 );
         cityLabels.Add( 15, label15 );
         cityLabels.Add( 16, label16 );
         cityLabels.Add( 17, label17 );
         cityLabels.Add( 18, label18 );
         cityLabels.Add( 19, label19 );
         cityLabels.Add( 20, label20 );
         cityLabels.Add( 21, label21 );
         cityLabels.Add( 22, label22 );
         cityLabels.Add( 23, label23 );
         cityLabels.Add( 24, label24 );
         cityLabels.Add( 25, label25 );
         cityLabels.Add( 26, label26 );
         cityLabels.Add( 27, label27 );
```

Fig. 21.22 Receiving temperature and weather data from a Web service. (Part 2 of 4.)

```
            cityLabels.Add( 28, label28 );
            cityLabels.Add( 29, label29 );
            cityLabels.Add( 30, label30 );
            cityLabels.Add( 31, label31 );
            cityLabels.Add( 32, label32 );
            cityLabels.Add( 33, label33 );
            cityLabels.Add( 34, label34 );

            // create Hashtable and populate with
            // all weather conditions
            Hashtable weather = new Hashtable();
            weather.Add( "SUNNY", "sunny" );
            weather.Add( "PTCLDY", "pcloudy" );
            weather.Add( "CLOUDY", "mcloudy" );
            weather.Add( "MOCLDY", "mcloudy" );
            weather.Add( "TSTRMS", "rain" );
            weather.Add( "RAIN", "rain" );
            weather.Add( "SNOW", "snow" );
            weather.Add( "VRYHOT", "vryhot" );
            weather.Add( "FAIR", "fair" );
            weather.Add( "RNSNOW", "rnsnow" );
            weather.Add( "SHWRS", "showers" );
            weather.Add( "WINDY", "windy" );
            weather.Add( "NOINFO", "noinfo" );
            weather.Add( "MISG", "noinfo" );
            weather.Add( "DRZL", "rain" );
            weather.Add( "HAZE", "noinfo" );
            weather.Add( "SMOKE", "mcloudy" );

            Bitmap background = new Bitmap( "images/back.png" );
            Font font = new Font( "Courier New", 8,
               FontStyle.Bold );

            // for every city
            for ( int i = 0; i < cities.Length; i++ )
            {
               // use Hashtable cityLabels to find the next Label
               Label currentCity = ( Label )cityLabels[ i + 1 ];

               // set current Label's image to image
               // corresponding to the city's weather condition -
               // find correct image name in Hashtable weather
               currentCity.Image = new Bitmap( "images/" +
                  weather[ descriptions[ i ].Trim() ] + ".png" );

               // set background image, font and forecolor
               // of Label
               currentCity.BackgroundImage = background;
               currentCity.Font = font;
               currentCity.ForeColor = Color.White;

```

Fig. 21.22 Receiving temperature and weather data from a Web service. (Part 3 of 4.)

```
            // set label's text to city name
            currentCity.Text = "\r\n" + cities[ i ] + " " +
               temperatures[ i ];
         }

      } // end of constructor

      // Visual Studio .NET generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new Client() );
      }

   } // end class Client

} // end namespace TemperatureClient
```



THE WEATHER FETCHER

CITY	HI/LO	CONDITIONS
ALBANY NY	39/46	
ATLANTA	47/70	
BOSTON	43/47	
BURLINGTON VT	37/43	
CHARLOTTE	43/67	
CLEVELAND	33/50	
DENVER	40/72	
GREAT FALLS	42/62	
HONOLULU	75/85	
KANSAS CITY	51/70	
LOS ANGELES	60/79	
MPLS ST PAUL	36/70	
NEW YORK CITY	45/53	
OKLAHOMA CITY	56/71	
PHILADELPHIA	44/53	
PITTSBURGH	35/50	
PORTLAND OR	44/54	

THE WEATHER FETCHER

CITY	HI/LO	CONDITIONS
ANCHORAGE	17/25	
ATLANTIC CITY	47/54	
BUFFALO	37/44	
CHARLESTON WV	34/56	
CHICAGO	35/59	
DALLAS FT WORTH	54/75	
DETROIT	34/50	
HARTFORD SPGFLD	40/47	
HOUSTON INTCNTL	54/78	
LAS VEGAS	59/78	
MIAMI BEACH	77/82	WINDY
NEW ORLEANS	60/76	
NORFOLK VA	49/58	
ORLANDO	67/75	WINDY
PHOENIX	71/83	
PORTLAND ME	37/44	
RENO	36/70	

Fig. 21.22 Receiving temperature and weather data from a Web service. (Part 4 of 4.)

Lines 60–63 of the constructor instantiate a **`TemperatureServer`** object, create a new **`CookieContainer`** object and update the weather data by calling method **`UpdateWeatherConditions`**. Lines 65–67 call **`TemperatureServer`** methods **`Cities`**, **`Descriptions`** and **`Temperatures`** to retrieve the city's weather and description information. Because the application presents weather data for so many cities, we must establish a way to organize the information in the **`Label`**s and to ensure that each weather description is accompanied by an appropriate image. To address these concerns, the program uses class **`Hashtable`** (discussed further in Chapter 23, Data Structures and Collections) to store all the **`Label`**s and weather descriptions and the names of their corresponding images. A **`Hashtable`** stores key-value pairs, in which both the key and the value can be any type of object. Method **`Add`** adds key-value pairs to a **`Hashtable`**. The class also provides an indexer to return the key value on which the **`Hashtable`** is indexed. Line 75 creates a **`Hashtable`** object, and lines 76–109 add the **`Label`**s to the **`Hashtable`**, using the numbers **`1`** through **`36`** as keys. Then, line 113 creates a second **`Hashtable`** object (**`weather`**) to contain pairs of weather conditions and the images associated with those conditions. Note that a given weather description does not necessarily correspond to the name of the PNG file containing the correct image. For example, both "**`TSTRMS`**" and "**`RAIN`**" weather conditions use the **`rain.png`** file.

Lines 137–157 set each **`Label`** so that it contains a city name, the current temperature in the city and an image corresponding to the weather condition for that city. Line 140 uses the **`Hashtable`** indexer to retrieve the next **`Label`** by passing as an argument the current value of **`i`** plus **`1`**. We add **`1`** because the **`Hashtable`** indexer begins at 0, despite the fact that both the labels and the **`Hashtable`** keys are numbered from 1–36.

Lines 145–146 set the **`Label`**'s image to the PNG image that corresponds to the city's weather condition. The application does this by retrieving the name of the PNG image from the **`weather Hashtable`**. The program eliminates any spaces in the description **`string`** by calling **`string`** method **`Trim`**. Lines 150–156 set several **`Label`**s' properties to achieve the visual effect seen in the output. For each label, we specify a blue-and-black background image (line 150). Lines 155–156 set each label's text so that it displays the correct information for each city (i.e., the city's name and temperature).

21.8 User-Defined Types in Web Services

The Web service discussed in the previous section returns arrays of **`string`**s. It would be much more convenient if **`TemperatureServer`** could return an array of **`CityWeather`** objects, instead of an array of **`string`**s. Fortunately, it is possible to define and employ user-defined types (also known as custom types) in a Web service. These types can be passed into or returned from Web-service methods. Web-service clients also can use these user-defined types, because the proxy class created for the client contains these type definitions. There are, however, some subtleties to keep in mind when using user-defined types in Web services; we point these out as we encounter them in the next example.

The case study in this section presents a math-tutoring program. The Web service generates random equations of type **`Equation`**. The client inputs information about the kind of mathematical example that the user wants (addition, subtraction or multiplication) and the skill level of the user (1 creates equations using one-digit numbers, 2 specifies more difficult equations involving two-digit numbers and 3 specifies the most difficult equations, containing three-digit numbers). It then generates an equation consisting of random numbers that have the proper number of digits. The client receives the **Equation** and uses a Windows Form to display the sample questions to the user.

We mentioned earlier that all data types passed to and from Web services must be supported by SOAP. How, then, can SOAP support a type that is not even created yet? In Chapter 17, Files and Streams, we discussed the serializing of data types, which enables them to be written to files. Similarly, custom types that are sent to or from a Web service are serialized, enabling them to be passed in XML format. This process is referred to as *XML serialization*.

When defining objects to be returned from Web-service methods, there are several subtleties to understand. For example, any object returned by a Web-service method must have a default constructor. Although all objects can be instantiated using a default **public** constructor (even if this constructor is not defined explicitly), a class returned from a Web service must have an explicitly defined constructor, even if its body is empty.

Common Programming Error 21.3

Failure to define explicitly a ***public*** *constructor for a type being used in a Web service results in a run-time error.*

A few additional requirements apply to custom types in Web services. Any variables of our user-defined type that we wish to access on the client-side must be declared **public**. We also must define both the **get** and **set** accessors of any properties that we wish to access at runtime. The Web service needs to have a way both to retrieve and manipulate such properties, because objects of the user-defined type will be converted into XML (when the objects are serialized) then converted back to objects (when they are de-serialized). During serialization, the property value must be read (through the **get** accessor); during de-serialization, the property value of the new object must be set (through the **set** accessor). If only one accessor is present, the client application will not have access to the property.

Common Programming Error 21.4

Defining only the ***get*** *or* ***set*** *accessor of a property for a user-defined type being used in a Web service results in a property that is inaccessible to the client.*

Common Programming Error 21.5

Clients of a Web service can access only that service's ***public*** *members. To allow access to* ***private*** *data, the programmer should provide* ***public*** *properties.*

Figure 21.23 displays class **Equation**. The constructor that is called (lines 18–37) takes three arguments—two integers representing the left and right operands and a **string** representing the algebraic operation to carry out. We define a default constructor (line 13–15) that calls another constructor (lines 18–37) and passes some default values. The constructor sets the **left**, **right** and **operation** fields, then calculates the appropriate result. We do not use this default constructor, but it must be defined in the program.

Class **Equation** defines properties **LeftHandSide**, **RightHandSide**, **Left**, **Right**, **Operation** and **Result**. The program does not need to modify the values of some of these properties, but implementation for the **set** accessor must be provided. **LeftHandSide** returns a **string** representing everything to the left of the "**=**" sign, and **RightHandSide** returns a **string** representing everything to the right of the "**=**" sign. **Left** returns the **int** to the left of the operator (known as the left operand), and **Right** returns the **int** to the right of the operator (known as the right operand). **Result** returns

```
// Fig. 21.23: Equation.cs
// Class Equation that contains
// information about an equation.

using System;

public class Equation
{
   private int left, right, result;
   private string operation;

   // required default constructor
   public Equation() : this( 0, 0, "+" )
   {
   }

   // constructor for class Equation
   public Equation( int leftValue, int rightValue,
      string operationType )
   {
      Left = leftValue;
      Right = rightValue;
      Operation = operationType;

      switch ( operationType )
      {
         case "+":
            Result = Left + Right;
            break;
         case "-":
            Result = Left - Right;
            break;
         case "*":
            Result = Left * Right;
            break;
      }
   }

   public override string ToString()
   {
      return Left.ToString() + " " + Operation + " " +
         Right.ToString() + " = " + Result.ToString();
   }

   // property returning string representing
   // left-hand side
   public string LeftHandSide
   {
      get
      {
         return Left.ToString() + " " + Operation + " " +
            Right.ToString();
      }
```

Fig. 21.23 Class that stores equation information. (Part 1 of 3.)

```

         set
         {
         }
      }

      // property returning string representing
      // right-hand side
      public string RightHandSide
      {
         get
         {
            return Result.ToString();
         }

         set
         {
         }
      }

      // left operand get and set property
      public int Left
      {
         get
         {
            return left;
         }

         set
         {
            left = value;
         }
      }

      // right operand get and set property
      public int Right
      {
         get
         {
            return right;
         }

         set
         {
            right = value;
         }
      }

      // get and set property of result of applying
      // operation to left and right operands
      public int Result
      {
```

Fig. 21.23 Class that stores equation information. (Part 2 of 3.)

```
         get
         {
            return result;
         }

         set
         {
            result = value;
         }
      }

      // get and set property for operation
      public string Operation
      {
         get
         {
            return operation;
         }

         set
         {
            operation = value;
         }
      }

   } // end class Equation
```

Fig. 21.23 Class that stores equation information. (Part 3 of 3.)

the answer to the equation, and **`Operation`** returns the operator. The program does not actually need the **`RightHandSide`** property, but we have chosen to include it in case other clients choose to use it. Figure 21.24 presents the **`Generator`** Web service that creates random, customized **`Equation`**s.

```
// Fig. 21.24: Generator.asmx.cs
// Web Service to generate random equations based on a
// specified operation and difficulty level.

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace EquationGenerator
{
   [ WebService( Namespace = "http://www.deitel.com/csphtp1/ch21",
      Description = "A Web service that generates questions " +
      "based on the specified mathematical operation and " +
      "level of difficulty chosen." ) ]
```

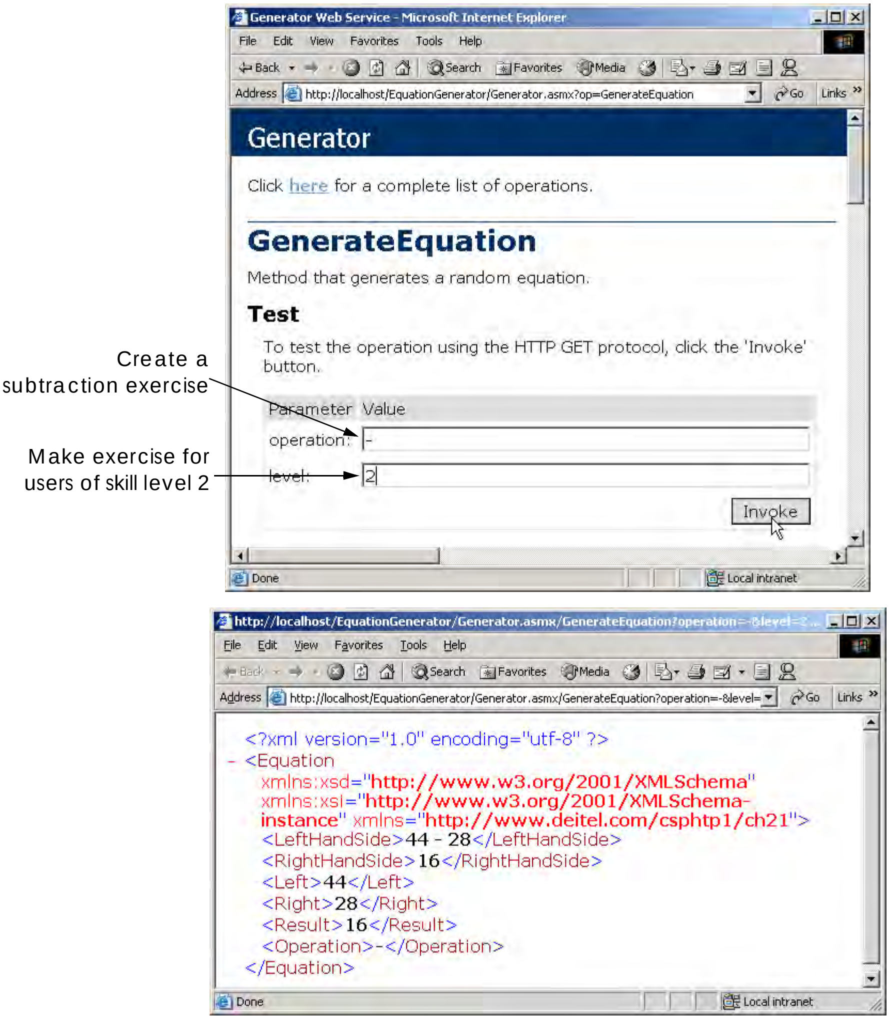
Fig. 21.24 Web service that generates random equations. (Part 1 of 2.)

```
19     public class Generator : System.Web.Services.WebService
20     {
21
22        // Visual Studio .NET generated code
23
24        [ WebMethod ( Description =
25           "Method that generates a random equation." ) ]
26        public Equation GenerateEquation( string operation,
27           int level )
28        {
29           // find maximum and minimum number to be used
30           int maximum = ( int ) Math.Pow( 10, level ),
31              minimum = ( int ) Math.Pow( 10, level - 1 );
32
33           Random random = new Random();
34
35           // create equation consisting of two random numbers
36           // between minimum and maximum parameters
37           Equation equation = new Equation(
38              random.Next( minimum, maximum ),
39              random.Next( minimum, maximum ), operation );
40
41           return equation;
42
43        } // end method GenerateEquation
44
45     } // end class Generator
46
47  } // end namespace EquationGenerator
```

Fig. 21.24 Web service that generates random equations. (Part 2 of 2.)

Web service `Generator` contains only one method, `GenerateEquation`. This method takes as arguments a `string` representing the operation we wish to perform and an `integer` representing the desired difficulty level of the equation. Figure 21.25 demonstrates the result of executing a test call of this Web service. Notice that the return value from our Web-service method is marked up as XML. However, this example differs from previous ones in that the XML specifies the values for all `public` properties and fields of the object that is being returned. The return object has been serialized into XML. Our proxy class takes this return value and deserializes it into an object (containing the `public` data from the original object) that then is passed back to the client.

Lines 30–31 define the lower and upper bounds for the random numbers that the method generates. To set these limits, the program first calls `static` method `Pow` of class `Math`—this method raises its first argument to the power of its second argument. Integer `maximum` represents the upper bound for a randomly generated number. The program raises `10` to the power of the specified `level` argument and then passes this value as the upper bound. For instance, if `level` is `1`, `maximum` is `10`; if `level` is `2`; `maximum` is `100` and so on. Variable `minimum`'s value is determined by raising `10` to a power one less then `level`. This calculates the smallest number with `level` digits. If `level` is `2`, `minimum` is `10`; if `level` is `3`, `minimum` is `100` and so on.



Generator Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/EquationGenerator/Generator.asmx?op=GenerateEquation
Go Links
Done
Local intranet

Generator

Click here for a complete list of operations.

GenerateEquation

Method that generates a random equation.

Test

To test the operation using the HTTP GET protocol, click the 'Invoke' button.

Parameter	Value
operation:	-
level:	2

Invoke

Create a subtraction exercise

Make exercise for users of skill level 2

http://localhost/EquationGenerator/Generator.asmx/GenerateEquation?operation=-&level=2...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/EquationGenerator/Generator.asmx/GenerateEquation?operation=-&level=
Go Links
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <Equation
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns="http://www.deitel.com/csphtp1/ch21">
    <LeftHandSide>44 - 28</LeftHandSide>
    <RightHandSide>16</RightHandSide>
    <Left>44</Left>
    <Right>28</Right>
    <Result>16</Result>
    <Operation>-</Operation>
  </Equation>
```

Fig. 21.25 Returning an object from a Web-service method.

Lines 37–39 create a new **Equation** object. The program calls **Random** method **Next**, which returns an integer that is greater than or equal to a specified lower bound, but less than a specified upper bound. In this example, **Random** generates a left operand value that is greater than or equal to **minimum**, but less than **maximum** (i.e., a number with **level** digits). The right operand is another random number with the same characteristics. The operation passed to the **Equation** constructor is the **string operation** that was received by **GenerateEquation**. The new **Equation** object is returned.

Figure 21.26 lists the math-tutoring application that uses the **Generator** Web service. The application calls **Generator**'s **GenerateEquation** method to create an **Equation** object. The application then displays the left-hand side of the **Equation** and waits for user input. In this example, the program accesses both class **Generator** and class **`Equation`** from within the **`localhost`** namespace—both are placed in this namespace when the proxy is generated.

The math-tutoring application displays a question and waits for input. The default setting for the difficulty level is **1**, but the user can change this at any time by choosing a level from among the top row of **`RadioButton`**s. Clicking any of the level options invokes **`levelRadioButtons_Click`** (lines 110–120), which sets integer **`level`** to the level selected by the user. Although the default setting for the question type is **Addition**, the user also can change this at any time by selecting one of the bottom-row **`RadioButton`**s. Doing so invokes the **`operationRadioButtons_Click`** (lines 91–107) event handler, which sets **`string operation`** so that it contains the symbol corresponding to the user's selection.

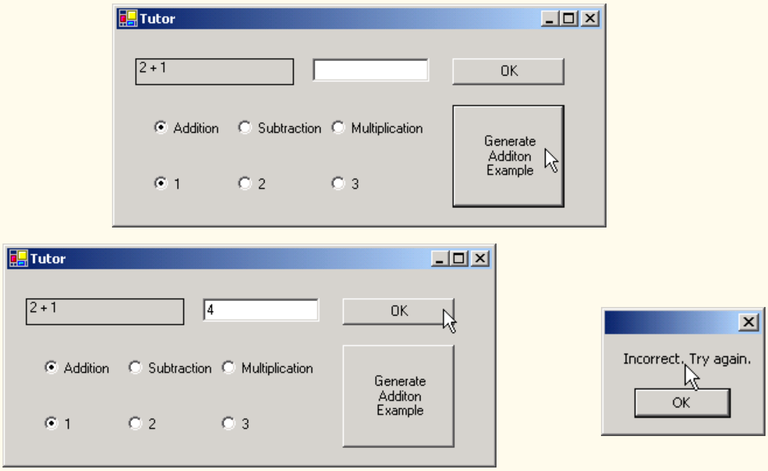
```
1  // Fig. 21.26: Tutor.cs
2  // Math tutor program.
3
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9
10 namespace EquationGeneratorClient
11 {
12    public class Tutor : System.Windows.Forms.Form
13    {
14       private System.Windows.Forms.Panel panel1;
15       private System.Windows.Forms.Panel panel2;
16
17       private System.Windows.Forms.Label questionLabel;
18       private System.Windows.Forms.TextBox answerTextBox;
19       private System.Windows.Forms.Button okButton;
20       private System.Windows.Forms.Button generateButton;
21
22       private System.Windows.Forms.RadioButton oneRadioButton;
23       private System.Windows.Forms.RadioButton twoRadioButton;
24       private System.Windows.Forms.RadioButton
25          threeRadioButton;
26       private System.Windows.Forms.RadioButton addRadioButton;
27       private System.Windows.Forms.RadioButton
28          subtractRadioButton;
29       private System.Windows.Forms.RadioButton
30          multiplyRadioButton;
31
32       private System.ComponentModel.Container components =
33          null;
34       private int level = 1;
35
36       private localhost.Equation equation;
37       private localhost.Generator generator =
38          new localhost.Generator();
39       private string operation = "+";
```

Fig. 21.26 Math tutor application. (Part 1 of 4.)

```

      // Visual Studio .NET generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new Tutor() );
      }

      // generates new equation on click event
      protected void generateButton_Click( object sender,
         System.EventArgs e )
      {
         // generate equation using current operation
         // and level
         equation = generator.GenerateEquation( operation,
            level );

         // display left-hand side of equation
         questionLabel.Text = equation.LeftHandSide;

         okButton.Enabled = true;
         answerTextBox.Enabled = true;

      } // end method generateButton_Click

      // check users answer
      protected void okButton_Click( object sender,
         System.EventArgs e )
      {
         // determine correct result from Equation
         // object
         int answer = equation.Result;

         // get user's answer
         int myAnswer = Int32.Parse( answerTextBox.Text );

         // test if user's answer is correct
         if ( answer == myAnswer )
         {
            questionLabel.Text = "";
            answerTextBox.Text = "";
            okButton.Enabled = false;
            MessageBox.Show( "Correct! Good job!" );
         }
         else
            MessageBox.Show( "Incorrect. Try again." );

      } // end method okButton_Click

```

Fig. 21.26 Math tutor application. (Part 2 of 4.)

```
      // set the selected operation
      protected void operationRadioButtons_Click( object sender,
         EventArgs e )
      {
         RadioButton item = ( RadioButton ) sender;

         // set the operation to be the appropriate symbol
         if ( item == addRadioButton )
            operation = "+";
         else if ( item == subtractRadioButton )
            operation = "-";
         else
            operation = "*";

         generateButton.Text = "Generate " + item.Text +
            " Example";

      } // end method operationRadioButtons_Click

      // set the current level
      protected void levelRadioButtons_Click( object sender,
         EventArgs e )
      {
         if ( sender == oneRadioButton )
            level = 1;
         else if ( sender == twoRadioButton )
            level = 2;
         else
            level = 3;

      } // end method levelRadioButtons_Click

   } // end class Tutor

} // end namespace EquationGeneratorClient
```

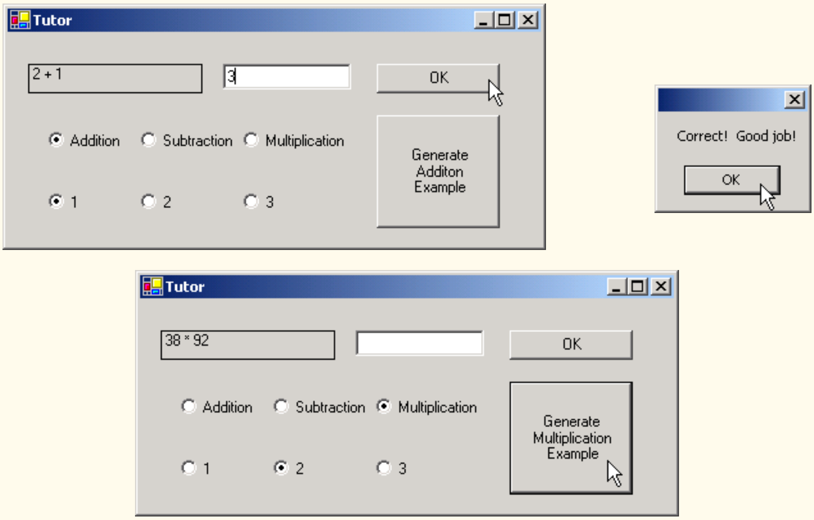


2 + 1

2 + 1 | 4

Incorrect. Try again.

Fig. 21.26 Math tutor application. (Part 3 of 4.)



Tutor

2 + 1

3

● Addition ○ Subtraction ○ Multiplication

● 1 ○ 2 ○ 3

Correct! Good job!

Tutor

38 * 92

○ Addition ○ Subtraction ● Multiplication

○ 1 ● 2 ○ 3

Fig. 21.26 Math tutor application. (Part 4 of 4.)

Event handler **`generateButton_Click`** (lines 50–64) invokes **`Generator`** method **`GenerateEquation`**. The left-hand side of the equation is displayed in **`questionLabel`** (line 59), and **`okButton`** is enabled so that the user can enter an answer. When the user clicks **OK**, **`okButton_Click`** (lines 67–88) checks whether the user provided the correct answer.

This chapter and the previous familiarized readers with the creation of Web applications and Web services, both of which enable users to request and receive data via the Internet. In the next chapter, we discuss the low-level details of how data are sent from one location to another (this process is called networking). Topics discussed in the next chapter include the implementation of servers and clients and the sending of data via sockets.

SUMMARY

- A Web service is an application that is stored on a remote machine and accessed through a remote procedure call.
- Distributed systems technologies enable applications to execute across multiple computers on a network.
- Web-services method calls are implemented using Simple Object Access Protocol (SOAP), an XML-based protocol describing how requests and responses are marked up so that they can be transferred via protocols such as HTTP.
- Methods are executed using a Remote Procedure Call (RPC). These methods are marked with the **`WebMethod`** attribute and are often referred to as Web-service methods.
- Requests to and responses from Web services are sent using SOAP by default. As long as a client can create and understand SOAP messages, the client can use Web services, regardless of the programming languages in which the Web services are written.
- A Web service in .NET has two parts: an ASMX file and a code-behind file.
- The ASMX file can be viewed in any Web browser and displays information about the Web service.

- The code-behind file contains the definition for the methods in the Web service.
- A service description is an XML document that conforms to the Web Service Description Language (WSDL).
- WSDL is an XML vocabulary that describes how Web services behave.
- The service description can be used by a client program to confirm the correctness of method calls at compile time.
- The ASMX file also provides a way for clients to execute test runs of the Web-service methods.
- SOAP, HTTP GET and HTTP POST are the three different ways of sending and receiving messages in Web services. The format used for these request and response messages is sometimes known as the wire protocol or wire format, because the format defines how information is sent "along the wire."
- The Simple Object Access Protocol (SOAP) is a platform-independent protocol that uses XML to make remote-procedure calls over HTTP.
- Requests to and responses from a Web-service method are packaged by default in a SOAP message—an XML message containing all the information necessary to process its contents.
- SOAP allows Web services to employ a variety of data types, including user-defined data types.
- When a program invokes a Web-service method, the request and all relevant information are packaged in a SOAP message and sent to the appropriate destination.
- When the Web service receives the SOAP message, it processes the message's contents, which specifies the method that the client wishes to execute and the arguments the client is passing to that method.
- When the Web service receives a request, the request is parsed, and the proper method is called with the specified arguments (if there are any). The response is sent back to the client as another SOAP message.
- An application that uses a Web service consists of two parts: a proxy class for the Web service and a client application that accesses the Web service via the proxy.
- A proxy class handles the task of transferring the arguments passed from the client into a SOAP message that is sent to the Web service. The proxy likewise handles the transferring of information in the SOAP response to the client.
- The **`Namespace`** property of a **`WebService`** attribute uniquely identifies a Web service.
- The **`Description`** property of a **`WebService`** attribute adds a description of the Web service when the Web service is displayed in a browser.
- Class **`WebService`** provides members that determine information about the user, the application and other topics relevant to the Web service.
- A Web service is not required to inherit from class **`WebService`**.
- A programmer specifies a method as a Web-service method by tagging it with the **`WebMethod`** attribute.
- Visual Studio provides a design view for each Web service, which allows the programmer to add components to the application.
- A proxy class is created from the Web service's WSDL file that enables the client to call Web-service methods over the Internet.
- Whenever a call is made in a client application to a Web-service method, a method in the proxy class is called. This method takes the method name and arguments passed by the client and formats them so that they can be sent as a request in a SOAP message.

- By default, the namespace of a proxy class is the name of the domain in which the Web service resides.
- UDDI is a project for developing a set of specifications that define how Web services should be discovered so that programmers searching for Web services can find them.
- A DISCO file is a file that specifies any Web services that are available in the current directory.
- There are two types of discovery files: Dynamic discovery files (**`.vsdisco`** extension) and static discovery files (**`.disco`** extension).
- Once a Web reference is created, a static discovery file is placed in the client's project. The static discovery file hard codes the locations of the ASMX and WSDL files.
- Dynamic discovery files are created so that a list of Web services is created when a client is searching for Web services.
- To store session information, the **`EnableSession`** property of the **`WebMethod`** attribute must be set to **`true`**.
- The use of session state in a Web service can make coding easier and reduce overhead.
- When storing session information, a Web service must have a way of identifying users between method calls. The approach is implemented using cookies, which are stored in a **`CookieContainer`**.
- Types can be defined by a programmer and used in a Web service. These types can be passed into or returned from Web-service methods.
- User-defined types can be sent to or returned from Web-service methods, because the types are defined in the proxy class created for the client.
- Custom types that are sent to or from a Web service are serialized as XML.
- Any object returned by a Web-service method must have a default constructor.
- Any variables of a custom type that we wish to make available to clients must be declared **`public`**.
- Properties of a custom type that we wish to make available to clients must have both **`get`** and **`set`** accessors defined.
- When an object is returned from a Web service, all its **`public`** properties and fields are marked up in XML. This information can then be transferred back into an object on the client side.

TERMINOLOGY

Add Web Reference dialog
ASMX file
ASP.NET Web Service project type
code-behind file in Web services
consuming a Web service
`CookieContainer` class
`CookieContainer` property
creating a proxy class for a Web service
`Description` property of a **`WebMethod`** attribute
`Description` property of a **`WebService`** attribute
`.disco` file extension
discovery (DISCO) files
distributed computing
distributed system
`EnableSession` property of a **`WebMethod`** attribute
exposing a Web-service method
firewall
Invoke button
`Namespace` property of a **`WebService`** attribute
`OpenRead` method of class **`WebClient`**
proxy class
publishing a Web service
remote machine
Remote Procedure Call (RPC)
session tracking in Web services
Simple Object Access Protocol (SOAP)
SOAP envelope
SOAP message

SOAP request
`System.Net`
Uniform Resource Locator (URL)
Universal Description, Discovery and Integration (UDDI)
`.vsdisco` file extension
Web service
Web Service Description Language (WSDL)
Web-service method
`WebClient` class
`WebMethod` attribute
`WebService` attribute
`WebService` class
wire format
wire protocol
XML serialization

SELF-REVIEW EXERCISES

21.1 State whether each of the following is *true* or *false*. If *false*, explain why.

- a) The purpose of a Web service is to create objects that are instantiated and used on the local machine.
- b) A Web server is required to create Web services and make them available.
- c) If a Web service is referenced by adding a Web reference to a client in Visual Studio .NET, a proxy class is not created.
- d) In .NET, a program communicating with a Web service uses HTTP GET by default to send and receive messages.
- e) A client can use only Web-service methods that are tagged with the `WebMethod` attribute.
- f) To enable session tracking in a Web-service method, the programmer sets the `EnableSession` property to `true` in the `WebMethod` attribute. No other action is required.
- g) An application can use only one Web service.
- h) Not all primitive data types can be returned from a Web service.
- i) `WebMethod`s methods cannot be declared `static`.
- j) A user-defined type used in a Web service must define both `get` and `set` accessors for any property that will be accessed in an application.

21.2 Fill in the blanks for each of the following statements:

- a) When messages are sent between an application and a Web service, each message is placed in a __________.
- b) A Web service can inherit from class __________.
- c) The class that defines a Web service usually is located in the __________ file for that Web service.
- d) The format used by a Web service to send and receive messages is usually known as the __________ or __________.
- e) A __________ file specifies any Web services that are available in the current directory.
- f) Class __________ is designed for interaction with resources identified by a URL.
- g) Web-service requests are sent over the Internet via the __________ protocol.
- h) To add a description for a Web service method in an ASMX page, the __________ property of the `WebService` attribute is used.
- i) Sending objects between a Web service and a client requires __________ of the object.
- j) A proxy class is defined in a namespace whose name is that of the __________ in which the Web service is defined.

ANSWERS TO SELF-REVIEW EXERCISES

21.1 a) False. Web services are used to execute methods on remote machines. The Web service receives the parameters it needs to execute a particular method, executes the method and then returns the result to the caller. b) True. c) True. d) False. A program communicating with a Web service uses

SOAP by default to send and receive messages. e) True. f) False. A **`CookieContainer`** also must be created on the client side. g) False. An application can use as many Web services as it needs. h) True. i) True. j) True.

21.2 a) SOAP message. b) **`WebService`**. c) code-behind. d) wire format, wire protocol. e) **`.disco`**. f) **`WebClient`**. g) HTTP. h) **`Description`**. i) XML serialization. j) domain.

EXERCISES

21.3 Create a Web service that stores phone-book entries in a database. Give the user the capability to enter new contacts and to find contacts by last name. Pass only primitive types as arguments to the Web service.

21.4 Modify Exercise 21.3 so that it uses a class named **`PhoneBookEntry`**. The client application should provide objects of type **`PhoneBookEntry`** to the Web service when adding contacts and should receive objects of type **`PhoneBookEntry`** when searching for contacts.

21.5 Modify the **`Blackjack`** Web service example in Section 21.5 to include a class **`Card`**. Have **`DealCard`** return an object of type **`Card`**. Also, have the client application keep track of what cards have been dealt, using **`Card`**s. Your card class should include properties to determine the face and suit of the card.

21.6 Modify the airline reservation example in Section 21.6 so that it contains two separate Web methods—one that allows users to view all available seats and another that allows users to reserve seats. Use an object of type **`Ticket`** to pass information to and from the Web service. This Web application should list all available seats in a **`ListBox`** and then allow the user to click a seat to reserve it. Your application must be able to handle cases where two users view available seats, one reserves a seat, and then the second user tries to reserve the same seat, not knowing that the database has changed since the page was loaded.

21.7 Modify the **`TemperatureServer`** example in Section 21.7 so that it returns an array of **`CityWeather`** objects that the client application uses to display the weather information.

21.8 Modify the Web service in the math-tutor example in Section 21.8 so that it includes a method that calculates how “close” the player is to the correct answer. The client application should provide the correct answer only after a user has offered numerous answers that were far from the correct one. Use your best judgment regarding what constitutes being “close” to the right answer. Remember that there should be a different formula for one-digit, two-digit and three-digit numbers. Also, give the program the capability of suggesting that users try a lower difficulty level if the users are consistently wrong.

22

Networking: Streams-Based Sockets and Datagrams

Objectives

- To be able to implement C# networking applications that use sockets and datagrams.
- To understand how to implement C# clients and servers that communicate with one another.
- To understand how to implement network-based collaborative applications.
- To construct a multithreaded server.

If the presence of electricity can be made visible in any part of a circuit, I see no reason why intelligence may not be transmitted instantaneously by electricity.
Samuel F. B. Morse

Mr. Watson, come here, I want you.
Alexander Graham Bell

What networks of railroads, highways and canals were in another age, the networks of telecommunications, information and computerization … are today.
Bruno Kreisky, Austrian Chancellor

Science may never come up with a better office-communication system than the coffee break.
Earl Wilson

Outline

22.1 Introduction
22.2 Establishing a Simple Server (Using Stream Sockets)
22.3 Establishing a Simple Client (Using Stream Sockets)
22.4 Client/Server Interaction with Stream-Socket Connections
22.5 Connectionless Client/Server Interaction with Datagrams
22.6 Client/Server Tic-Tac-Toe Using a Multithreaded Server

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

22.1 Introduction

The Internet and the World Wide Web have generated a great deal of excitement in the business and computing communities. The Internet ties the “information world” together; the Web makes the Internet easy to use while providing the flair of multimedia. Organizations see both the Internet and the Web as crucial to their information-systems strategies. C# and the .NET Framework offer a number of built-in networking capabilities that facilitate Internet-based and Web-based applications development. C# not only can specify parallelism through multithreading, but also can enable programs to search the Web for information and collaborate with programs running on other computers internationally.

In Chapters 20 and 21, we began our presentation of C#’s networking and distributed-computing capabilities. We discussed Web Forms and Web Services, two high-level networking technologies that enable programmers to develop distributed applications in C#. In this chapter, we focus on the networking technologies that support C#’s ASP.NET capabilities and can be used to build distributed applications.

Our discussion of networking focuses on both sides of a *client–server relationship*. The *client* requests that some action be performed; the *server* performs the action and responds to the client. A common implementation of this request–response model is between Web browsers and Web servers. When users select Web sites that they wish to view through a browser (the client application), the browser makes a request to the appropriate Web server (the server application). The server normally responds to the client by sending the appropriate HTML Web pages.

C#’s networking capabilities are grouped into several namespaces. The fundamental networking capabilities are defined by classes and interfaces of namespace **`System.Net.Sockets`**. Through this namespace, C# offers *socket-based communications*, which enable developers to view networking as if it were file I/O. This means that a program can read from a *socket* (network connection) or write to a socket as easily as it can read from or write to a file. Sockets are the fundamental way to perform network communications in the .NET Framework. The term “socket” refers to the Berkeley Sockets Interface, which was developed in 1978 for network programming with UNIX and was popularized by C and C++ programmers.

The classes and interfaces of namespace **`System.Net.Sockets`** also offer *packet-based communications*, through which individual *packets* of information are transmitted—

this is a common method of transmitting audio and video over the Internet. In this chapter, we show how to create and manipulate sockets and how to communicate via packets of data.

Socket-based communications in C# employ *stream sockets.* With stream sockets, a *process* (running program) establishes a *connection* to another process. While the connection is in place, data flows between the processes in continuous *streams*. For this reason, stream sockets are said to provide a *connection-oriented service*. The popular *TCP (Transmission Control Protocol)* facilitates stream-socket transmission.

By contrast, packet-based communications in C# employ *datagram sockets*, through which individual *packets* of information are transmitted. Unlike TCP, the protocol used to enable datagram sockets—*UDP*, the *User Datagram Protocol*—is a *connectionless service* and does not guarantee that packets will arrive in any particular order. In fact, packets can be lost or duplicated and can arrive out of sequence. Applications that use UDP often require significant extra programming to deal with these problems. UDP is most appropriate for network applications that do not require the error checking and reliability of TCP. For example, several online multi-player games use UDP, because speed is more important than perfect accuracy in these types of applications. Stream sockets and the TCP protocol will be the most desirable method of communication for the vast majority of C# programmers.

Performance Tip 22.1

Connectionless services generally offer better performance but less reliability than do connection-oriented services.

Portability Tip 22.1

The TCP protocol and its related set of protocols enable intercommunication among a wide variety of heterogeneous computer systems (i.e., computer systems with different processors and different operating systems).

22.2 Establishing a Simple Server (Using Stream Sockets)

Typically, with TCP and stream sockets, a server “waits” for a connection request from a client. Often, the server program contains a control structure or block of code that executes continuously until the server receives a request. On receiving a request, the server establishes a connection with the client. The server then uses this connection to handle future requests from that client and to send data to the client.

The establishment of a simple server with TCP and stream sockets in C# requires five steps. The first step is to create an object of class ***`TcpListener`***, which belongs to namespace **`System.Net.Sockets`**. This class represents a TCP stream socket through which a server can listen for requests. A call to the **`TcpListener`** constructor, such as

```
TcpListener server = new TcpListener( port );
```

binds (assigns) the server to the specified *port number*. A port number is a numeric identifier that a process uses to identify itself at a given *network address***,** also known as an *Internet Protocol Address* (*IP Address*). IP addresses identify computers on the Internet. In fact, Web-site names, such as **`www.deitel.com`**, are aliases for IP addresses. Any process that performs networking identifies itself via an *IP address/port number pair*. Hence, no two processes can have the same port number at a given IP address. The explicit binding of a socket to a port (using method ***`Bind`*** of class **`Socket`**) is usually unnecessary, because class `TcpListener` and other classes discussed in this chapter hide this binding (i.e., bind sockets to ports implicitly), plus they perform other socket-initialization operations.

Software Engineering Observation 22.1

Port numbers can have values between 0 and 65535. Many operating systems reserve port numbers below 1024 for system services (such as e-mail and Web servers). Applications must be granted special privileges to use these reserved port numbers. Usually, a server-side application should not specify port numbers below 1024 as connection ports, because some operating systems might reserve these numbers.

Common Programming Error 22.1

Attempting to bind an already assigned port at a given IP address is a logic error.

To receive requests, the `TcpListener` first must listen for them. The second step in our connection process is to call `TcpListener`'s ***`Start`*** method, which causes the `TcpListener` object to begin listening for connection requests. The third step establishes the connection between the server and client. The server listens indefinitely for a request—i.e., the execution of the server-side application waits until some client attempts to connect with it. The server creates a connection to the client upon receipt of a connection request. An object of class ***`System.Net.Sockets.Socket`*** manages each connection to the client. Method ***`AcceptSocket`*** of class `TcpListener` waits for a connection request, then creates a connection when a request is received. This method returns a `Socket` object upon connection, as in the statement

```
Socket connection = server.AcceptSocket();
```

When the server receives a request, method `AcceptSocket` calls method ***`Accept`*** of the `TcpListener`'s underlying `Socket` to make the connection. This is an example of C#'s hiding of networking complexity from the programmer. The programmer can write the preceding statement into a server-side program, then allow the classes of namespace `System.Net.Sockets` to handle the details of accepting requests and establishing connections.

Step four is the processing phase, in which the server and the client communicate via methods ***`Receive`*** and ***`Send`*** of class `Socket`. Note that these methods, as well as TCP and stream sockets, can be used only when the server and client are connected. By contrast, through `Socket` methods ***`SendTo`*** and ***`ReceiveFrom`***, UDP and datagram sockets can be used when no connection exists.

The fifth step is the connection-termination phase. When the client and server have finished communicating, the server uses method ***`Close`*** of the `Socket` object to close the connection. Most servers then return to step two (i.e., wait for another client's connection request).

One problem associated with the server scheme described in this section is that step four *blocks* other requests while processing a client's request, so that no other client can connect with the server while the code that defines the processing phase is executing. The most common technique for addressing this problem is to use multithreaded servers, which place the processing-phase code in a separate thread. When the server receives a connection request, the server *spawns*, or creates, a `Thread` to process the connection, leaving its `TcpListener` (or `Socket`) free to receive other connections.

Software Engineering Observation 22.2

Using C#'s multithreading capabilities, we can create servers that can manage simultaneous connections with multiple clients. This multithreaded-server architecture is precisely what popular UNIX and Windows network servers use.

Software Engineering Observation 22.3

A multithreaded server can be implemented to create a thread that manages network I/O across a reference to a ***`Socket`*** *object returned by method* ***`AcceptSocket`****. A multithreaded server also can be implemented to maintain a pool of threads that manage network I/O across newly created* ***`Socket`****s.*

Performance Tip 22.2

In high-performance systems with abundant memory, a multithreaded server can be implemented to create a pool of threads. These threads can be assigned quickly to handle network I/O across each multiple ***`Socket`****. Thus, when a connection is received, the server does not incur the overhead of thread creation.*

22.3 Establishing a Simple Client (Using Stream Sockets)

We create TCP-stream-socket clients via a process that requires four steps. In the first step, we create an object of class ***`TcpClient`*** (which belongs to namespace **`System.Net.Sockets`**) to connect to the server. This connection is established through method ***`Connect`*** of class **`TcpClient`**. One overloaded version of this method receives two arguments—the server's IP address and the port number—as in the following:

```
TcpClient client = new TcpClient();
client.Connect( serverAddress, serverPort );
```

Here, **`serverPort`** is an **`int`** that represents the server's port number; **`serverAddress`** can be either an ***`IPAddress`*** instance (that encapsulates the server's IP address) or a **`string`** that specifies the server's hostname. Alternatively, the programmer could pass an object reference of class ***`IPEndPoint`***, which represents an IP address/port number pair, to a different overload of method **`Connect`**. Method **`Connect`** of class **`TcpClient`** calls method ***`Connect`*** of class **`Socket`** to establish the connection. If the connection is successful, method **`TcpClient.Connect`** returns a positive integer; otherwise, it returns **`0`**.

In step two, the **`TcpClient`** uses its method ***`GetStream`*** to get a ***`NetworkStream`*** so that it can write to and read from the server. **`NetworkStream`** methods ***`WriteByte`*** and ***`Write`*** can be used to output individual bytes or sets of bytes to the server, respectively; similarly, **`NetworkStream`** methods ***`ReadByte`*** and ***`Read`*** can be used to input individual bytes or sets of bytes from the server, respectively.

The third step is the processing phase, in which the client and the server communicate. In this phase, the client uses methods **`Read`**, **`ReadByte`**, **`Write`** and **`WriteByte`** of class **`NetworkStream`** to perform the appropriate communications. Using a process similar to that used by servers, a client can employ threads to prevent blocking of communications with other servers while processing data from one connection.

After the transmission is complete, step four requires the client to close the connection by calling method ***`Close`*** of the **`NetworkStream`** object. This closes the underlying

`Socket` (if the **`NetworkStream`** has a reference to that **`Socket`**). Then, the client calls method ***`Close`*** of class **`TcpClient`** to terminate the TCP connection. At this point, a new connection can be established through method **`Connect`**, as we have described.

22.4 Client/Server Interaction with Stream-Socket Connections

The applications in Fig. 22.1 and Fig. 22.2 use the classes and techniques discussed in the previous two sections to construct a simple *client/server chat application*. The server waits for a client's request to make a connection. When a client application connects to the server, the server application sends an array of bytes to the client, indicating that the connection was successful. The client then displays a message notifying the user that a connection has been established.

Both the client and the server applications contain **`TextBox`**es that enable users to type messages and send them to the other application. When either the client or the server sends message "**`TERMINATE`**," the connection between the client and the server terminates. The server then waits for another client to request a connection. Figure 22.1 and Fig. 22.2 provide the code for classes **`Server`** and **`Client`**, respectively. Figure 22.2 also contains screen captures displaying the execution between the client and the server.

```
// Fig. 22.1: Server.cs
// Set up a Server that will receive a connection from a client,
// send a string to the client, and close the connection.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Threading;
using System.Net.Sockets;
using System.IO;

// server that awaits client connections (one at a time) and
// allows a conversation between client and server
public class Server : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox inputTextBox;
   private System.Windows.Forms.TextBox displayTextBox;
   private Socket connection;
   private Thread readThread;

   private System.ComponentModel.Container components = null;
   private NetworkStream socketStream;
   private BinaryWriter writer;
   private BinaryReader reader;

```

Fig. 22.1 Server portion of a client/server stream-socket connection. (Part 1 of 4.)

```
   // default constructor
   public Server()
   {
      InitializeComponent();

      // create a new thread from the server
      readThread = new Thread( new ThreadStart( RunServer ) );
      readThread.Start();
   }

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new Server() );
   }

   protected void Server_Closing(
      object sender, CancelEventArgs e )
   {
      System.Environment.Exit( System.Environment.ExitCode );
   }

   // sends the text typed at the server to the client
   protected void inputTextBox_KeyDown(
      object sender, KeyEventArgs e )
   {
      // sends the text to the client
      try
      {
         if ( e.KeyCode == Keys.Enter && connection != null )
         {
            writer.Write( "SERVER>>> " + inputTextBox.Text );

            displayTextBox.Text +=
               "\r\nSERVER>>> " + inputTextBox.Text;

            // if the user at the server signaled termination
            // sever the connection to the client
            if ( inputTextBox.Text == "TERMINATE" )
               connection.Close();

            inputTextBox.Clear();
         }
      }
      catch ( SocketException )
      {
         displayTextBox.Text += "\nError writing object";
      }
   } // inputTextBox_KeyDown

```

Fig. 22.1 Server portion of a client/server stream-socket connection. (Part 2 of 4.)

```
      // allows a client to connect and displays the text it sends
      public void RunServer()
      {
         TcpListener listener;
         int counter = 1;

         // wait for a client connection and display the text
         // that the client sends
         try
         {
            // Step 1: create TcpListener
            listener = new TcpListener( 5000 );

            // Step 2: TcpListener waits for connection request
            listener.Start();

            // Step 3: establish connection upon client request
            while ( true )
            {
               displayTextBox.Text = "Waiting for connection\r\n";

               // accept an incoming connection
               connection = listener.AcceptSocket();

               // create NetworkStream object associated with socket
               socketStream = new NetworkStream( connection );

               // create objects for transferring data across stream
               writer = new BinaryWriter( socketStream );
               reader = new BinaryReader( socketStream );

               displayTextBox.Text += "Connection " + counter +
                  " received.\r\n";

               // inform client that connection was successfull
               writer.Write( "SERVER>>> Connection successful" );

               inputTextBox.ReadOnly = false;
               string theReply = "";

               // Step 4: read String data sent from client
               do
               {
                  try
                  {
                     // read the string sent to the server
                     theReply = reader.ReadString();

                     // display the message
                     displayTextBox.Text += "\r\n" + theReply;
                  }

```

Fig. 22.1 Server portion of a client/server stream-socket connection. (Part 3 of 4.)

```
                  // handle exception if error reading data
                  catch ( Exception )
                  {
                     break;
                  }

               } while ( theReply != "CLIENT>>> TERMINATE"  &&
                  connection.Connected );

               displayTextBox.Text +=
                  "\r\nUser terminated connection";

               // Step 5: close connection
               inputTextBox.ReadOnly = true;
               writer.Close();
               reader.Close();
               socketStream.Close();
               connection.Close();

               ++counter;
            }
         } // end try

         catch ( Exception error )
         {
            MessageBox.Show( error.ToString() );
         }

      } // end method RunServer

   } // end class Server
```

Fig. 22.1 Server portion of a client/server stream-socket connection. (Part 4 of 4.)

```
// Fig. 22.2: Client.cs
// Set up a Client that will read information sent from a Server
// and display the information.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Threading;
using System.Net.Sockets;
using System.IO;

// connects to a chat server
public class Client : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox inputTextBox;
```

Fig. 22.2 Client portion of a client/server stream-socket connection. (Part 1 of 5.)

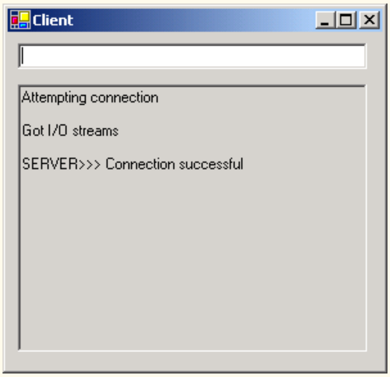
```
      private System.Windows.Forms.TextBox displayTextBox;

      private NetworkStream output;
      private BinaryWriter writer;
      private BinaryReader reader;

      private string message = "";

      private Thread readThread;

      private System.ComponentModel.Container components = null;

      // default constructor
      public Client()
      {
         InitializeComponent();

         readThread = new Thread( new ThreadStart( RunClient ) );
         readThread.Start();
      }

      // Visual Studio .NET-generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new Client() );
      }

      protected void Client_Closing(
         object sender, CancelEventArgs e )
      {
         System.Environment.Exit( System.Environment.ExitCode );
      }

      // sends text the user typed to server
      protected void inputTextBox_KeyDown (
         object sender, KeyEventArgs e )
      {
         try
         {
            if ( e.KeyCode == Keys.Enter )
            {
               writer.Write( "CLIENT>>> " + inputTextBox.Text );

               displayTextBox.Text +=
                  "\r\nCLIENT>>> " + inputTextBox.Text;

               inputTextBox.Clear();
            }
         }
```

Fig. 22.2 Client portion of a client/server stream-socket connection. (Part 2 of 5.)

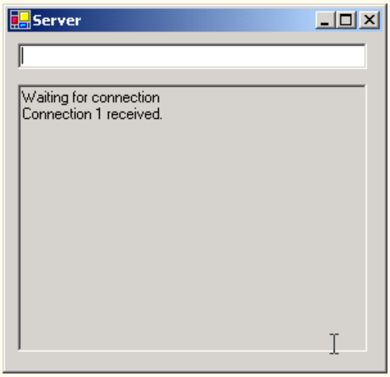
```
         catch ( SocketException ioe )
         {
            displayTextBox.Text += "\nError writing object";
         }

      } // end method inputTextBox_KeyDown

      // connect to server and display server-generated text
      public void RunClient()
      {
         TcpClient client;

         // instantiate TcpClient for sending data to server
         try
         {
            displayTextBox.Text += "Attempting connection\r\n";

            // Step 1: create TcpClient and connect to server
            client = new TcpClient();
            client.Connect( "localhost", 5000 );

            // Step 2: get NetworkStream associated with TcpClient
            output = client.GetStream();

            // create objects for writing and reading across stream
            writer = new BinaryWriter( output );
            reader = new BinaryReader( output );

            displayTextBox.Text += "\r\nGot I/O streams\r\n";

            inputTextBox.ReadOnly = false;

            // loop until server signals termination
            do
            {

               // Step 3: processing phase
               try
               {
                  // read message from server
                  message = reader.ReadString();
                  displayTextBox.Text += "\r\n" + message;
               }

               // handle exception if error in reading server data
               catch ( Exception )
               {
                  System.Environment.Exit(
                     System.Environment.ExitCode );
               }
            } while( message != "SERVER>>> TERMINATE" );

```

Fig. 22.2 Client portion of a client/server stream-socket connection. (Part 3 of 5.)

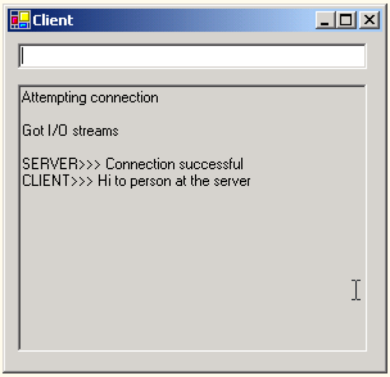
```
         displayTextBox.Text += "\r\nClosing connection.\r\n";

         // Step 4: close connection
         writer.Close();
         reader.Close();
         output.Close();
         client.Close();
         Application.Exit();
      }

      // handle exception if error in establishing connection
      catch ( Exception error )
      {
         MessageBox.Show( error.ToString() );
      }

   } // end method RunClient

} // end class Client
```



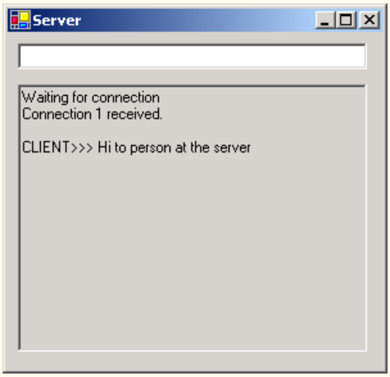
```
Attempting connection

Got I/O streams

SERVER>>> Connection successful
```



```
Waiting for connection
Connection 1 received.
```

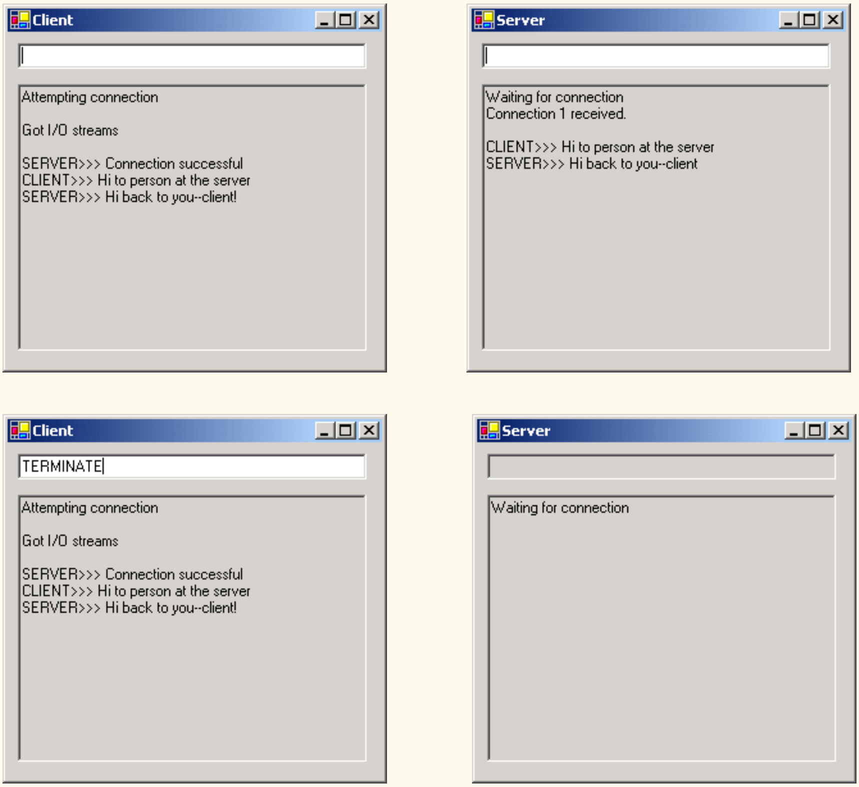


```
Attempting connection

Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hi to person at the server
```



```
Waiting for connection
Connection 1 received.

CLIENT>>> Hi to person at the server
```

Fig. 22.2 Client portion of a client/server stream-socket connection. (Part 4 of 5.)



```
Attempting connection

Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hi to person at the server
SERVER>>> Hi back to you--client!
```

```
Waiting for connection
Connection 1 received.

CLIENT>>> Hi to person at the server
SERVER>>> Hi back to you--client
```

```
TERMINATE
```

```
Attempting connection

Got I/O streams

SERVER>>> Connection successful
CLIENT>>> Hi to person at the server
SERVER>>> Hi back to you--client!
```

```
Waiting for connection
```

Fig. 22.2 Client portion of a client/server stream-socket connection. (Part 5 of 5.)

As we analyze this example, we begin by discussing class **`Server`** (Fig. 22.1). In the constructor, line 34 creates a **`Thread`** that will accept connections from clients. Line 35 starts the **`Thread`**, which invokes method **`RunServer`** (lines 81–160). Method **`RunServer`** initializes the server to receive connection requests and process connections. Line 91 instantiates the **`TcpListener`** to listen for a connection request from a client at port **`5000`** (Step 1). Line 94 then calls method **`Start`** of the **`TcpListener`** object, which causes the **`TcpListener`** to begin waiting for requests (Step 2).

Lines 97–152 declare an infinite **`while`** loop that establishes connections requested by clients (Step 3). Line 102 calls method **`AcceptSocket`** of the **`TcpListener`** object, which returns a **`Socket`** upon successful connection. The thread in which method **`AcceptSocket`** is called stops executing until a connection is established. The **`Socket`** object will manage the connection. Line 105 passes this **`Socket`** object as an argument to the constructor of a **`NetworkStream`** object. Class **`NetworkStream`** provides access to streams across a network—in this example, the **`NetworkStream`** object provides access to the **`Socket`** connection. Lines 108–109 create instances of the ***`BinaryWriter`*** and ***`BinaryReader`*** classes for writing and reading data. We pass the **`NetworkStream`** object as an argument to each constructor; **`BinaryWriter`** can write bytes to the **`NetworkStream`**, and **`BinaryReader`** can read bytes from **`NetworkStream`**. Lines 111–112 append text to the **`TextBox`**, indicating that a connection was received.

`BinaryWriter` method ***`Write`*** has many overloaded versions, which enable the method to write various types to a stream. (You might remember that we used these overloaded methods in Chapter 17 to write record data to files.) Line 115 uses method **`Write`** to send to the client a **`string`** notifying the user of a successful connection. Lines 121–139 declare a **`do`**/**`while`** structure that executes until the server receives a message indicating connection termination (i.e., **`CLIENT>>> TERMINATE`**). Line 126 uses **`BinaryReader`** method ***`ReadString`*** to read a **`string`** from the stream (Step 4). (You might remember that we also used this method in Chapter 17 to read records' first-name and last-name **`string`**s from files.) Method **`ReadString`** blocks until a **`string`** is read. To prevent the whole server from blocking, we use a separate **`Thread`** to handle the transfer of information. The **`while`** statement loops until there is more information to read—this results in I/O blocking, which causes the program always to appear frozen. However, if we run this portion of the program in a separate **`Thread`**, the user can interact with the Windows **`Form`** and send messages while the program waits in the background for incoming messages.

When the chat is complete, lines 146–149 close the **`BinaryWriter`**, **`BinaryReader`**, **`NetworkStream`** and **`Socket`** (Step 5) by invoking their respective **`Close`** methods. The server then waits for another client connection request by returning to the beginning of the **`while`** loop (line 97).

When the user of the server application enters a **`string`** in the **`TextBox`** and presses the *Enter* key, event handler **`inputTextBox_KeyDown`** (lines 53–78) reads the **`string`** and sends it via method **`Write`** of class **`BinaryWriter`**. If a user terminates the server application, line 69 calls method **`Close`** of the **`Socket`** object to close the connection.

Lines 46–50 define the **`Server_Closing`** event handler for the **`Closing`** event. The event closes the application and uses **`System.Environment.Exit`** method with parameter **`System.Environment.ExitCode`** to terminate all threads. Method **`Exit`** of class **`Environment`** closes all threads associated with the application.

Figure 22.2 lists the code for the **`Client`** object. Like the **`Server`** object, the **`Client`** object creates a **`Thread`** (lines 35–36) in its constructor to handle all incoming messages. **`Client`** method **`RunClient`** (lines 77–137) connects to the **`Server`**, receives data from the **`Server`** and sends data to the **`Server`** (when the user presses *Enter*). Lines 87–88 instantiate a **`TcpClient`** object, then call its method **`Connect`** to establish a connection (Step 1). The first argument to method **`Connect`** is the name of the server—in our case, the server's name is ***`"localhost"`***, meaning that the server is located on the same machine as the client. The **`localhost`** is also known as the *loopback IP address* and is equivalent to the IP address ***`127.0.0.1`***. This value sends the data transmission back to the sender's IP address. [*Note*: We chose to demonstrate the client/server relationship by connecting between programs that are executing on the same computer (**`localhost`**). Normally, this argument would contain the Internet address of another computer.] The second argument to method **`Connect`** is the server port number. This number must match the port number at which the server waits for connections.

The **`Client`** uses a **`NetworkStream`** to send data to and receive data from the server. The client obtains the **`NetworkStream`** on line 91 through a call to **`TcpClient`** method **`GetStream`** (Step 2). The **`do`**/**`while`** structure in lines 102–119 loops until the client receives the connection-termination message (**`SERVER>>> TERMINATE`**). Line 109 uses **`BinaryReader`** method **`ReadString`** to obtain the next message from the server (Step

3). Line 110 displays the message, and lines 124–127 close the **`BinaryWriter`**, **`BinaryReader`**, **`NetworkStream`** and **`TcpClient`** objects (Step 4).

When the user of the client application enters a **`string`** in the **`TextBox`** and presses the *Enter* key, the event handler **`inputTextBox_KeyDown`** (lines 54–74) reads the **`string`** from the **`TextBox`** and sends it via **`BinaryWriter`** method **`Write`**. Notice that, here, the **`Server`** receives a connection, processes it, closes it and waits for the next one. In a real-world application, a server would likely receive a connection, set up the connection to be processed as a separate thread of execution and wait for new connections. The separate threads that process existing connections can continue to execute while the **`Server`** concentrates on new connection requests.

22.5 Connectionless Client/Server Interaction with Datagrams

Up to this point, we have discussed connection-oriented, streams-based transmission. Now, we consider connectionless transmission using datagrams.

Connection-oriented transmission is similar to interaction over a telephone system, in which a user dials a number and is *connected* to the telephone of the party they wish to connect. The system maintains the connection for the duration of the phone call, regardless of whether the users are speaking.

By contrast, connectionless transmission via *datagrams* more closely resembles the method by which the postal service carries and delivers mail. Connectionless transmission bundles and sends information in *packets* called datagrams, which can be thought of as similar to posted letters. If a large message will not fit in one envelope, that message is broken into separate message pieces and placed in separate, sequentially numbered envelopes. All the letters are mailed at once. The letters might arrive in order, out of order or not at all. The person at the receiving end reassembles the message pieces into sequential order before attempting to interpret the message. If the message is small enough to fit in one envelope, the sequencing problem is eliminated, but it is still possible that the message will never arrive. (Unlike with posted mail, duplicate of datagrams could reach receiving computers.) C# provides the ***`UdpClient`*** class for connectionless transmission. Like **`TcpListener`** and **`TcpClient`**, **`UdpClient`** uses methods from class **`Socket`**. The **`UdpClient`** methods ***`Send`*** and ***`Receive`*** are used to transmit data with **`Socket`**'s **`SendTo`** method and to read data with **`Socket`**'s **`ReceiveFrom`** method, respectively.

The programs in Fig. 22.3 and Fig. 22.4 use datagrams to send *packets* of information between a client and server applications. In the **`Client`** application, the user types a message into a **`TextBox`** and presses *Enter*. The client converts the message to a **`byte`** array and sends it to the server. The server receives the packet and displays the packet's information, then *echoes*, or returns, the packet back to the client. When the client receives the packet, the client displays the packet's information. In this example, the implementations of the **`Client`** and **`Server`** classes are similar.

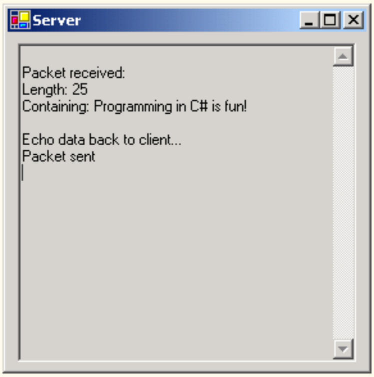
```
// Fig. 22.5: Server.cs
// Set up a Server that will receive packets from a
// client and send packets to a client.

```

Fig. 22.3 Server-side portion of connectionless client/server computing. (Part 1 of 3.)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Net;
using System.Net.Sockets;
using System.Threading;

// create the UDP server
public class Server : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox displayTextBox;
   private UdpClient client;
   private IPEndPoint receivePoint;
   private System.ComponentModel.Container components = null;

   // no-argument constructor
   public Server()
   {
      InitializeComponent();

      client = new UdpClient( 5000 );
      receivePoint = new IPEndPoint( new IPAddress( 0 ), 0 );
      Thread readThread = new Thread(
         new ThreadStart( WaitForPackets ) );

      readThread.Start();
   }

   // Visual Studio .NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new Server() );
   }

   // shut down the server
   protected void Server_Closing(
      object sender, CancelEventArgs e )
   {
      System.Environment.Exit( System.Environment.ExitCode );
   }

   // wait for a packet to arrive
   public void WaitForPackets()
   {
      while ( true )
      {
         // receive byte array from client
         byte[] data = client.Receive( ref receivePoint );
```

Fig. 22.3 Server-side portion of connectionless client/server computing. (Part 2 of 3.)

```

            // output packet data to TextBox
            displayTextBox.Text += "\r\nPacket received:" +
               "\r\nLength: " + data.Length + "\r\nContaining: " +
               System.Text.Encoding.ASCII.GetString( data );

            displayTextBox.Text +=
               "\r\n\r\nEcho data back to client...";

            // echo information from packet back to client
            client.Send( data, data.Length, receivePoint );
            displayTextBox.Text += "\r\nPacket sent\r\n";
         }

      } // end method WaitForPackets

   } // end class Server
```



```

Packet received:
Length: 25
Containing: Programming in C# is fun!

Echo data back to client...
Packet sent
```

Fig. 22.3 Server-side portion of connectionless client/server computing. (Part 3 of 3.)

```
// Fig. 22.6: Client.cs
// Set up a Client that sends packets to a server and receives
// packets from a server.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Net;
using System.Net.Sockets;
using System.Threading;

```

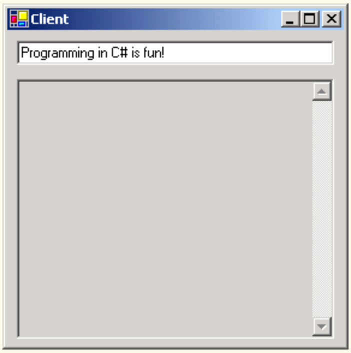
Fig. 22.4 Client portion of connectionless client/server computing. (Part 1 of 3.)

```
// run the UDP client
public class Client : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox inputTextBox;
   private System.Windows.Forms.TextBox displayTextBox;

   private UdpClient client;
   private IPEndPoint receivePoint;

   private System.ComponentModel.Container components = null;

   // no-argument constructor
   public Client()
   {
      InitializeComponent();

      receivePoint = new IPEndPoint( new IPAddress( 0 ), 0 );
      client = new UdpClient( 5001 );
      Thread thread =
         new Thread( new ThreadStart( WaitForPackets ) );
      thread.Start();
   }

   // Visual Studio.NET generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new Client() );
   }

   // shut down the client
   protected void Client_Closing(
      object sender, CancelEventArgs e )
   {
      System.Environment.Exit( System.Environment.ExitCode );
   }

   // send a packet
   protected void inputTextBox_KeyDown(
      object sender, KeyEventArgs e )
   {
      if ( e.KeyCode == Keys.Enter )
      {
         // create packet (datagram) as string
         string packet = inputTextBox.Text;
         displayTextBox.Text +=
            "\r\nSending packet containing: " + packet;

         // convert packet to byte array
         byte[] data =
            System.Text.Encoding.ASCII.GetBytes( packet );

```

Fig. 22.4 Client portion of connectionless client/server computing. (Part 2 of 3.)

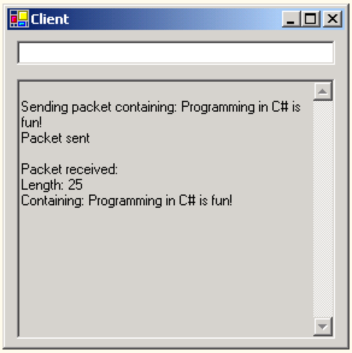
```
         // send packet to server on port 5000
         client.Send( data, data.Length, "localhost", 5000 );
         displayTextBox.Text += "\r\nPacket sent\r\n";
         inputTextBox.Clear();
      }
   } // end method inputTextBox_KeyDown

   // wait for packets to arrive
   public void WaitForPackets()
   {
      while ( true )
      {
         // receive byte array from server
         byte[] data = client.Receive( ref receivePoint );

         // output packet data to TextBox
         displayTextBox.Text += "\r\nPacket received:" +
            "\r\nLength: " + data.Length + "\r\nContaining: " +
            System.Text.Encoding.ASCII.GetString( data ) +
            "\r\n";
      }

   } // end method WaitForPackets

} // end class Client
```

`Client` window before sending a packet to the server



Programming in C# is fun!

`Client` window after sending a packet to the server and receiving it back



```
Sending packet containing: Programming in C# is
fun!
Packet sent

Packet received:
Length: 25
Containing: Programming in C# is fun!
```

Fig. 22.4 Client portion of connectionless client/server computing. (Part 3 of 3.)

The code in Fig. 22.3 defines the **`Server`** for this application. Line 28 in the constructor for class **`Server`** creates an instance of the **`UdpClient`** class that receives data at port **`5000`**. This initializes the underlying **`Socket`** for communications. Line 29 creates an instance of class **`IPEndPoint`** to hold the IP address and port number of the client(s) that transmit to **`Server`**. The first argument to the constructor of **`IPEndPoint`** is an **`IPAddress`** object; the second argument to the constructor for **`IPEndPoint`** is the port number of the endpoint. These values are both **`0`**, because we need only instantiate an empty **`IPEndPoint`** object. The IP addresses and port numbers of clients are copied into the **`IPEndPoint`** when datagrams are received from clients.

`Server` method `WaitForPackets` (lines 52–72) executes an infinite loop while waiting for data to arrive at the `Server`. When information arrives, the `UdpClient` method `Receive` (line 57) receives a `byte` array from the client. We include `Receive` in the `IPEndPoint` object created in the constructor; this provides the method with a reference to an `IPEndPoint` into which the program copies the client's IP address and port number. This program will compile and run without an exception even if the reference to the `IPEndPoint` object is `null`, because method `Receive` initializes the `IPEndPoint` if it is `null`.

Good Programming Practice 22.1

Initialize all references to objects (to a value other than `null`*). This protects code from methods that do not check their parameters for* `null` *references.*

Lines 60–65 update the `Server`'s display to include the packet's information and content. Line 68 echoes the data back to the client, using `UdpClient` method `Send`. This version of `Send` takes three arguments: The byte array to send, an `int` representing the array's length and the `IPEndPoint` to which to send the data. We use array `data` returned by method `Receive` as the data, the length of array `data` as the length and the `IPEndPoint` passed to method `Receive` as the data's destination. The IP address and port number of the client that sent the data to `Server` are stored in `receivePoint`, so merely passing `receivePoint` to `Send` allows `Server` to respond to the client.

Class `Client` (Fig. 22.4) works similarly to class `Server`, except that the `Client` object sends packets only when the user types a message in a `TextBox` and presses the *Enter* key. When this occurs, the program calls event handler `inputTextBox_KeyDown` (lines 54–73). Lines 65–66 convert the `string` that the user entered in the `TextBox` to a `byte` array. Line 69 calls `UdpClient` method `Send` to send the `byte` array to the `Server` that is located on `localhost` (i.e., the same machine). We specify the port as `5000`, which we know to be `Server`'s port.

Line 32 instantiates a `UdpClient` object to receive packets at port `5001`—we choose port `5001`, because the `Server` already occupies port `5000`. Method `WaitForPackets` of class `Client` (lines 76–90) uses an infinite loop to wait for these packets. The `UdpClient` method `Receive` blocks until a packet of data is received (line 81). The blocking performed by method `Receive` does not prevent class `Client` from performing other services (e.g., handling user input), because a separate thread runs method `WaitForPackets`.

When a packet arrives, lines 84–87 display its contents in the `TextBox`. The user can type information into the `Client` window's `TextBox` and press the *Enter* key at any time, even while a packet is being received. The event handler for the `TextBox` processes the event and sends the data to the server.

22.6 Client/Server Tic-Tac-Toe Using a Multithreaded Server

In this section, we present our capstone networking example—the popular game Tic-Tac-Toe, implemented with stream sockets and client/server techniques. The program consists of a `Server` application (Fig. 22.5) and two `Client` applications (Fig. 22.6); `Server` allows the `Client`s to connect to the server and play Tic-Tac-Toe. We depict the output in Fig. 22.7. When the server receives a client connection, lines 72–83 of

Fig. 22.5 create an instance of class **`Player`** to process the client in a separate thread of execution. This enables the server to handle requests from both clients. The server assigns value **`"X"`** to the first client that connects (player **`X`** makes the first move), then assigns value **`"O"`** to the second client. Throughout the game, the server maintains information regarding the status of the board so that the server can validate players' requested moves. However, neither the server nor the client can establish whether a player has won the game—in this application, method **`GameOver`** (lines 143–147) always returns **`false`**. Exercise 22.7 asks the reader to implement functionality that enables the application to determine a winner. Each **`Client`** maintains its own GUI version of the Tic-Tac-Toe board to display the game. The clients can place marks only in empty squares on the board. Class **`Square`** (Fig. 22.7) is used to define squares on the Tic-Tac-Toe board.

```
1  // Fig. 22.5: Server.cs
2  // This class maintains a game of Tic-Tac-Toe for two
3  // client applications.
4
5  using System;
6  using System.Drawing;
7  using System.Collections;
8  using System.ComponentModel;
9  using System.Windows.Forms;
10 using System.Data;
11 using System.Net.Sockets;
12 using System.Threading;
13 using System.IO;
14
15 // awaits connections from two clients and allows them to
16 // play tic-tac-toe against each other
17 public class Server : System.Windows.Forms.Form
18 {
19    private System.Windows.Forms.TextBox displayTextBox;
20
21    private byte[] board;
22
23    private Player[] players;
24    private Thread[] playerThreads;
25
26    private TcpListener listener;
27    private int currentPlayer;
28    private Thread getPlayers;
29
30    private System.ComponentModel.Container components = null;
31
32    internal bool disconnected = false;
33
34    // default constructor
35    public Server()
36    {
37       InitializeComponent();
38
39       board = new byte[ 9 ];
```

Fig. 22.5 Server side of client/server Tic-Tac-Toe program. (Part 1 of 6.)

```

         players = new Player[ 2 ];
         playerThreads = new Thread[ 2 ];
         currentPlayer = 0;

         // accept connections on a different thread
         getPlayers = new Thread( new ThreadStart( SetUp ) );
         getPlayers.Start();
      }

      // Visual Studio .NET-generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new Server() );
      }

      protected void Server_Closing(
         object sender, CancelEventArgs e )
      {
         disconnected = true;
      }

      // accepts connections from 2 players
      public void SetUp()
      {
         // set up Socket
         listener = new TcpListener( 5000 );
         listener.Start();

         // accept first player and start a thread for him or her
         players[ 0 ] =
            new Player( listener.AcceptSocket(), this, 0 );
         playerThreads[ 0 ] = new Thread(
            new ThreadStart( players[ 0 ].Run ) );
         playerThreads[ 0 ].Start();

         // accept second player and start a thread for him or her
         players[ 1 ] =
            new Player( listener.AcceptSocket(), this, 1 );
         playerThreads[ 1 ] =
            new Thread( new ThreadStart( players[ 1 ].Run ) );
         playerThreads[ 1 ].Start();

         // let the first player know that the other player has
         // connected
         lock ( players[ 0 ] )
         {
            players[ 0 ].threadSuspended = false;
            Monitor.Pulse( players[ 0 ] );
         }
      } // end method SetUp
```

Fig. 22.5 Server side of client/server Tic-Tac-Toe program. (Part 2 of 6.)

```

   // appends the argument to text in displayTextBox
   public void Display( string message )
   {
      displayTextBox.Text += message + "\r\n";
   }

   // determine if a move is valid
   public bool ValidMove( int location, int player )
   {
      // prevent another thread from making a move
      lock ( this )
      {
         // while it is not the current player's turn, wait
         while ( player != currentPlayer )
            Monitor.Wait( this );

         // if the desired square is not occupied
         if ( !IsOccupied( location ) )
         {
            // set the board to contain the current player's mark
            board[ location ] = ( byte ) ( currentPlayer == 0 ?
               'X' : 'O' );

            // set the currentPlayer to be the other player
            currentPlayer = ( currentPlayer + 1 ) % 2;

            // notify the other player of the move
            players[ currentPlayer ].OtherPlayerMoved( location );

            // alert the other player it's time to move
            Monitor.Pulse( this );

            return true;
         }
         else
            return false;
      }
   } // end method ValidMove

   // determines whether the specified square is occupied
   public bool IsOccupied( int location )
   {
      if ( board[ location ] == 'X' || board[ location ] == 'O' )
         return true;
      else
         return false;
   }

   // determines if the game is over
   public bool GameOver()
   {
```

Fig. 22.5 Server side of client/server Tic-Tac-Toe program. (Part 3 of 6.)

```
         // place code here to test for a winner of the game
         return false;
      }

   } // end class Server

   public class Player
   {
      internal Socket connection;
      private NetworkStream socketStream;
      private Server server;
      private BinaryWriter writer;
      private BinaryReader reader;

      private int number;
      private char mark;
      internal bool threadSuspended = true;

      // constructor requiring Socket, Server and int objects
      // as arguments
     public Player( Socket socket, Server serverValue, int newNumber )
      {
         mark = ( newNumber == 0 ? 'X' : 'O' );

         connection = socket;

         server = serverValue;
         number = newNumber;

         // create NetworkStream object for Socket
         socketStream = new NetworkStream( connection );

         // create Streams for reading/writing bytes
         writer = new BinaryWriter( socketStream );
         reader = new BinaryReader( socketStream );

      } // end constructor

      // signal other player of move
      public void OtherPlayerMoved( int location )
      {
         // signal that opponent moved
         writer.Write( "Opponent moved" );
         writer.Write( location ); // send location of move
      }

      // allows the players to make moves and receives moves
      // from other player
      public void Run()
      {
         bool done = false;

```

Fig. 22.5 Server side of client/server Tic-Tac-Toe program. (Part 4 of 6.)

```
         // display on the server that a connection was made
         server.Display( "Player " + ( number == 0 ? 'X' : 'O' )
            + " connected" );

         // send the current player's mark to the server
         writer.Write( mark );

         // if number equals 0 then this player is X, so send
         writer.Write( "Player " + ( number == 0 ?
            "X connected\r\n" : "O connected, please wait\r\n" ) );

         // wait for another player to arrive
         if ( mark == 'X' )
         {
            writer.Write( "Waiting for another player" );

            // wait for notification from server that another
            // player has connected
            lock ( this )
            {
               while ( threadSuspended )
                  Monitor.Wait( this );
            }

            writer.Write( "Other player connected. Your move" );

         } // end if

         // play game
         while ( !done )
         {
            // wait for data to become available
            while ( connection.Available == 0 )
            {
               Thread.Sleep( 1000 );

               if ( server.disconnected )
                 return;
            }

            // receive data
            int location = reader.ReadInt32();

            // if the move is valid, display the move on the
            // server and signal the move is valid
            if ( server.ValidMove( location, number ) )
            {
               server.Display( "loc: " + location );
               writer.Write( "Valid move." );
            }

```

Fig. 22.5 Server side of client/server Tic-Tac-Toe program. (Part 5 of 6.)

```
248          // signal the move is invalid
249          else
250             writer.Write( "Invalid move, try again" );
251
252          // if game is over, set done to true to exit while loop
253          if ( server.GameOver() )
254             done = true;
255
256       } // end while loop
257
258       // close the socket connection
259       writer.Close();
260       reader.Close();
261       socketStream.Close();
262       connection.Close();
263
264    } // end method Run
265
266 } // end class Player
```

Fig. 22.5 Server side of client/server Tic-Tac-Toe program. (Part 6 of 6.)

`Server` (Fig. 22.5) uses its constructor (lines 35–48) to create a **`byte`** array to store the moves the players have made (line 39). The program creates an array of two references to **`Player`** objects (line 41) and an array of two references to **`Thread`** objects (line 42). Each element in both arrays corresponds to a Tic-Tac-Toe player. Variable **`currentPlayer`** is set to **`0`**, which corresponds to player **`"X."`** In our program, player **`"X"`** makes the first move (line 43). Lines 46–47 create and start **`Thread getPlayers`**, which the **`Server`** uses to accept connections so that the current **`Thread`** does not block while awaiting players.

Thread **`getPlayers`** executes method **`SetUp`** (lines 65–92), which creates a **`TcpListener`** object to listen for requests on port **`5000`** (lines 68–69). This object then listens for connection requests from the first and second players. Lines 72–73 and 79–80 instantiate **`Player`** objects representing the players, and lines 74–75 and 81–82 create two **`Thread`**s that execute the **`Run`** methods of each **`Player`** object.

The **`Player`** constructor (Fig. 22.5, lines 165–181) receives as arguments a reference to the **`Socket`** object (i.e., the connection to the client), a reference to the **`Server`** object and an **`int`** indicating the mark (**`"X"`** or **`"O"`**) used by that player. In this case study, **`Server`** calls method **`Run`** (lines 193–264) after instantiating a **`Player`** object. Lines 198–206 notify the server of a successful connection and send to the client the **`char`** that the client will place on the board when making a move. If **`Run`** is executing for **`Player "X"`**, lines 211–221 execute, causing **`Player "X"`** to wait for a second player to connect. Lines 217–218 define a **`while`** loop that suspends the **`Player "X" Thread`** until the server signals that **`Player "O"`** has connected. The server notifies the **`Player`** of the connection by setting the **`Player`**’s **`threadSuspended`** variable to **`false`** (line 89). When **`threadSuspended`** becomes **`false`**, **`Player`** exits the **`while`** loop of lines 217–218.

Method **`Run`** executes the **`while`** structure (lines 226–256), enabling the user to play the game. Each iteration of this structure waits for the client to send an **`int`** specifying where on the board to place the **`"X"`** or **`"O"`**—the **`Player`** then places the mark on the board, if the specified mark location is valid (e.g., that location does not already contain a mark). Note that the **`while`** structure continues execution only if **`bool`** variable **`done`** is **`false`**. This variable is set to **`true`** by event handler **`Server_Closing`** of class **`Server`**, which is invoked when the server closes the connection.

```
// Fig. 22.6: Client.cs
// Client for the TicTacToe program.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Net.Sockets;
using System.Threading;
using System.IO;

// represents a tic-tac-toe player
public class Client : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label idLabel;

   private System.Windows.Forms.TextBox displayTextBox;

   private System.Windows.Forms.Panel panel1;
   private System.Windows.Forms.Panel panel2;
   private System.Windows.Forms.Panel panel3;
   private System.Windows.Forms.Panel panel5;
   private System.Windows.Forms.Panel panel6;
   private System.Windows.Forms.Panel panel4;
   private System.Windows.Forms.Panel panel7;
   private System.Windows.Forms.Panel panel8;
   private System.Windows.Forms.Panel panel9;

   private Square[ , ] board;
   private Square currentSquare;

   private Thread outputThread;

   private TcpClient connection;
   private NetworkStream stream;
   private BinaryWriter writer;
   private BinaryReader reader;

   private char myMark;
   private bool myTurn;

   private SolidBrush brush;
   private System.ComponentModel.Container components = null;

   bool done = false;
```

Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 1 of 7.)

```

      // default constructor
      public Client()
      {
         InitializeComponent();

         board = new Square[ 3, 3 ];

         // create 9 Square objects and place them on the board
         board[ 0, 0 ] = new Square( panel1, ' ', 0 );
         board[ 0, 1 ] = new Square( panel2, ' ', 1 );
         board[ 0, 2 ] = new Square( panel3, ' ', 2 );
         board[ 1, 0 ] = new Square( panel4, ' ', 3 );
         board[ 1, 1 ] = new Square( panel5, ' ', 4 );
         board[ 1, 2 ] = new Square( panel6, ' ', 5 );
         board[ 2, 0 ] = new Square( panel7, ' ', 6 );
         board[ 2, 1 ] = new Square( panel8, ' ', 7 );
         board[ 2, 2 ] = new Square( panel9, ' ', 8 );

         // create a SolidBrush for writing on the Squares
         brush = new SolidBrush( Color.Black );

         // Make connection to sever and get the associated
         // network stream. Start separate thread to allow this
         // program to continually update its output in textbox.
         connection = new TcpClient( "localhost", 5000 );
         stream = connection.GetStream();

         writer = new BinaryWriter( stream );
         reader = new BinaryReader( stream );

         // start a new thread for sending and receiving messages
         outputThread = new Thread( new ThreadStart( Run ) );
         outputThread.Start();
      } // end Client constructor

      // Visual Studio .NET-generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new Client() );
      }

      protected void Client_Paint (
         object sender, System.Windows.Forms.PaintEventArgs e )
      {
         PaintSquares();
      }

```

Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 2 of 7.)

```
   protected void Client_Closing(
      object sender, CancelEventArgs e )
   {
      done = true;
   }

   // draws the mark of each square
   public void PaintSquares()
   {
      Graphics g;

      // draw the appropriate mark on each panel
      for ( int row = 0; row < 3; row++ )
         for ( int column = 0; column < 3; column++ )
         {
            // get the Graphics for each Panel
            g = board[ row, column ].SquarePanel.CreateGraphics();

            // draw the appropriate letter on the panel
            g.DrawString( board[ row, column ].Mark.ToString(),
               this.Font, brush, 8, 8 );
         }
   } // end method PaintSquares

   // send location of the clicked square to server
   protected void square_MouseUp(
      object sender, System.Windows.Forms.MouseEventArgs e )
   {
      // for each square check if that square was clicked
      for ( int row = 0; row < 3; row++ )
         for ( int column = 0; column < 3; column++ )
            if ( board[ row, column ].SquarePanel == sender )
            {
               CurrentSquare = board[ row, column ];

               // send the move to the server
               SendClickedSquare( board[ row, column ].Location );
            }
   } // end method square_MouseUp

   // control thread that allows continuous update of the
   // textbox display
   public void Run()
   {
      // first get players's mark (X or O)
      myMark = reader.ReadChar();
      idLabel.Text = "You are player \"" + myMark + "\"";
      myTurn = ( myMark == 'X' ? true : false );

      // process incoming messages
      try
      {
```

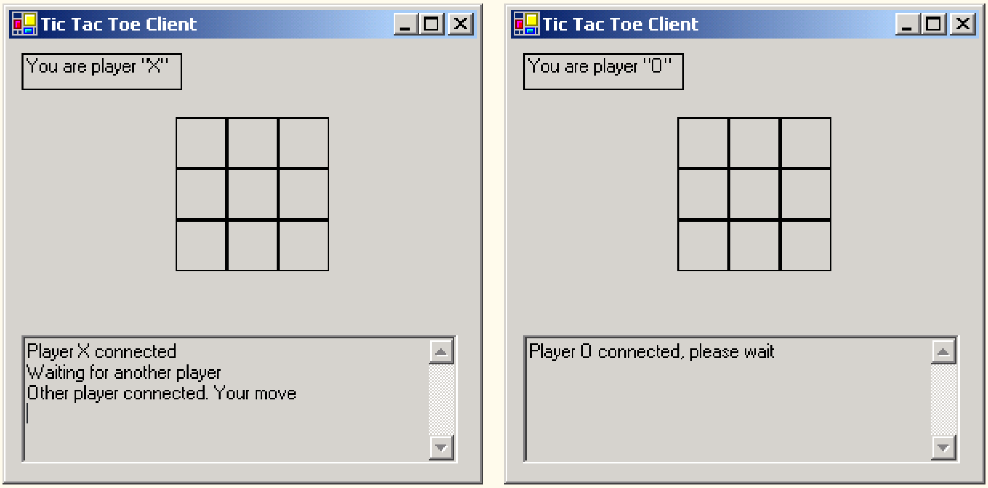
Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 3 of 7.)

```
         // receive messages sent to client
         while ( true )
            ProcessMessage( reader.ReadString() );
       }
       catch ( EndOfStreamException )
       {
          MessageBox.Show( "Server is down, game over", "Error",
             MessageBoxButtons.OK, MessageBoxIcon.Error );
       }

   } // end method Run

   // process messages sent to client
   public void ProcessMessage( string message )
   {
      // if the move player sent to the server is valid
      // update the display, set that square's mark to be
      // the mark of the current player and repaint the board
      if ( message == "Valid move." )
      {
         displayTextBox.Text +=
            "Valid move, please wait.\r\n";
         currentSquare.Mark = myMark;
         PaintSquares();
      }

      // if the move is invalid, display that and it is now
      // this player's turn again
      else if ( message == "Invalid move, try again" )
      {
         displayTextBox.Text += message + "\r\n";
         myTurn = true;
      }

      // if opponent moved
      else if ( message == "Opponent moved" )
      {
         // find location of their move
         int location = reader.ReadInt32();

         // set that square to have the opponents mark and
         // repaint the board
         board[ location / 3, location % 3 ].Mark =
            ( myMark == 'X' ? 'O' : 'X' );
         PaintSquares();

         displayTextBox.Text +=
            "Opponent moved.  Your turn.\r\n";

         // it is now this player's turn
         myTurn = true;
      }

```

Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 4 of 7.)

```
         // display the message
         else
            displayTextBox.Text += message + "\r\n";

      } // end method ProcessMessage

      // sends the server the number of the clicked square
      public void SendClickedSquare( int location )
      {
         // if it is the current player's move right now
         if ( myTurn )
         {
            // send the location of the move to the server
            writer.Write( location );

            // it is now the other player's turn
            myTurn = false;
         }
      }

      // write-only property for the current square
      public Square CurrentSquare
      {
         set
         {
            currentSquare = value;
         }
      }

   } // end class Client
```

1.



You are player "X"

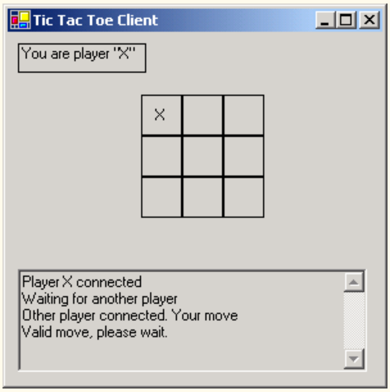
Player X connected
Waiting for another player
Other player connected. Your move

You are player "O"

Player O connected, please wait

Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 5 of 7.)

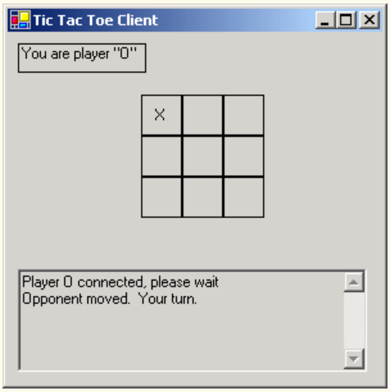
2.


Tic Tac Toe Client

You are player "X"

X		

```
Player X connected
Waiting for another player
Other player connected. Your move
Valid move, please wait.
```

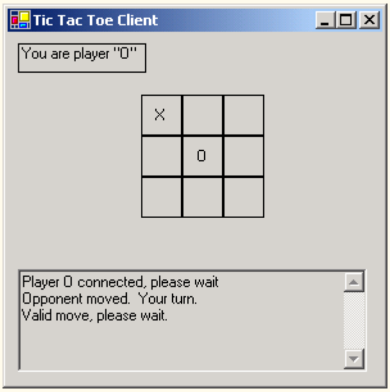

Tic Tac Toe Client

You are player "O"

X		

```
Player O connected, please wait
Opponent moved.  Your turn.
```

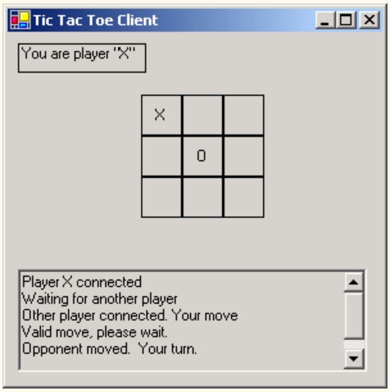
3.


Tic Tac Toe Client

You are player "O"

X		
	O	

```
Player O connected, please wait
Opponent moved.  Your turn.
Valid move, please wait.
```

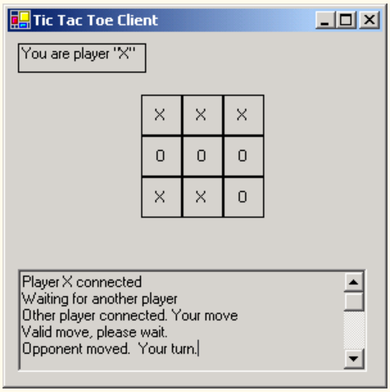

Tic Tac Toe Client

You are player "X"

X		
	O	

```
Player X connected
Waiting for another player
Other player connected. Your move
Valid move, please wait.
Opponent moved.  Your turn.
```

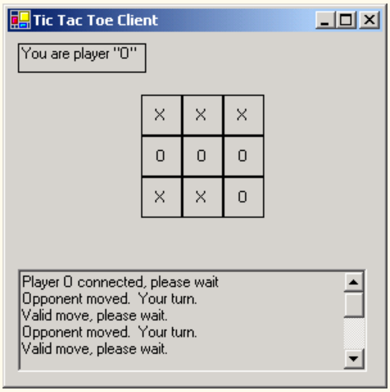
4.


Tic Tac Toe Client

You are player "X"

X	X	X
O	O	O
X	X	O

```
Player X connected
Waiting for another player
Other player connected. Your move
Valid move, please wait.
Opponent moved.  Your turn.
```

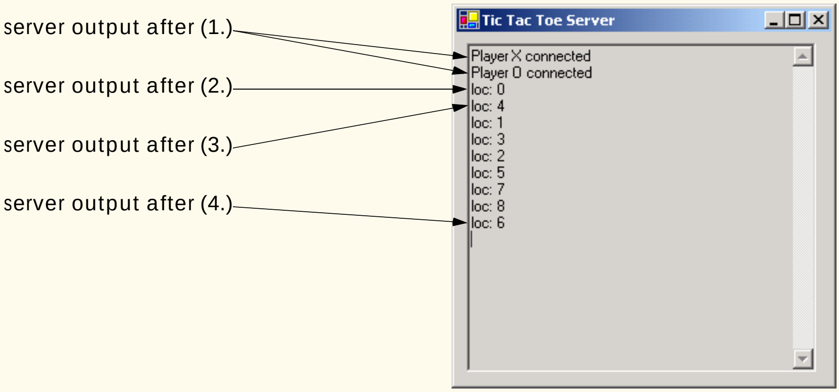

Tic Tac Toe Client

You are player "O"

X	X	X
O	O	O
X	X	O

```
Player O connected, please wait
Opponent moved.  Your turn.
Valid move, please wait.
Opponent moved.  Your turn.
Valid move, please wait.
```

Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 6 of 7.)



```
Player X connected
Player O connected
loc: 0
loc: 4
loc: 1
loc: 3
loc: 2
loc: 5
loc: 7
loc: 8
loc: 6
```

Fig. 22.6 Client side of client/server Tic-Tac-Toe program. (Part 7 of 7.)

Line 229 of Fig. 22.5 begins a **`while`** that loops until **`Socket`** property **`Available`** indicates that there is information to receive from the **`Socket`** (or until the server disconnects from the client). If there is no information, the thread goes to sleep for one second. Upon awakening, the thread uses property **`Disconnected`** to check for whether server variable **`disconnect`** is **`true`**. If the value is **`true`**, the **`Thread`** exits the method (thus terminating the **`Thread`**); otherwise, the **`Thread`** loops again. However, if property **`Available`** indicates that there is data to receive, the **`while`** loop of lines 229–235 terminates, enabling the information to be processed.

This information contains an **`int`** representing the location in which the client wants to place a mark. Line 238 calls method **`ReadInt32`** of the **`BinaryReader`** object (which reads from the **`NetworkStream`** created with the **`Socket`**) to read this **`int`**. Line 242 then passes the **`int`** to **`Server`** method **`ValidMove`**. If this method validates the move, the **`Player`** places the mark in the desired location.

Method **`ValidMove`** (lines 101–131) sends to the client a message indicating whether the move was valid. Locations on the board correspond to numbers from **`0`**–**`8`** (**`0`**–**`2`** for the first row, **`3`**–**`5`** for the second and **`6`**–**`8`** for the third). All statements in method **`ValidMove`** are enclosed in a **`lock`** statement that allows only one move to be attempted at a time. This prevents two players from modifying the game's state information simultaneously. If the **`Player`** attempting to validate a move is not the current player (i.e., the one allowed to make a move), that **`Player`** is placed in a *wait* state until it is that **`Player`**'s turn to move. If the user attempts to place a mark on a location that already contains a mark, method **`ValidMove`** returns **`false`**. However, if the user has selected an unoccupied location (line 111), lines 114–115 place the mark on the local representation of the board. Line 121 notifies the other **`Player`** that a move has been made, and line 124 invokes the **`Pulse`** method so that the waiting **`Player`** can validate a move. The method then returns **`true`** to indicate that the move is valid.

When a **`Client`** application (Fig. 22.6) executes, it creates a **`TextBox`** to display messages from the server and the Tic-Tac-Toe board representation. The board is created out of nine **`Square`** objects (Fig. 22.7) that contain **`Panel`**s on which the user can click, indicating the position on the board in which to place a mark. The **`Client`**'s constructor (line 50–82) opens a connection to the server (line 73) and obtains a reference to the connection's associated **`NetworkStream`** object from **`TcpClient`** (line 74). Lines 80–81 start a thread to read messages sent from the server to the client. The server passes messages (for example, whether each move is valid) to method **`ProcessMessage`** (lines 163–207). If the message indicates that a move is valid (line 168), the client sets its mark to the current square (the square that the user clicked) and repaints the board. If the message indicates that a move is invalid (line 178), the client notifies the user to click a different square. If the message indicates that the opponent made a move (line 185), line 188 reads from the server an **`int`** specifying where on the board the client should place the opponent's mark.

```
// Fig. 22.7: Square.cs
// A Square on the TicTacToe board.

using System.Windows.Forms;

// the representation of a square in a tic-tac-toe grid
public class Square
{
   private Panel panel;
   private char mark;
   private int location;

   // constructor
   public Square( Panel newPanel, char newMark, int newLocation )
   {
      panel = newPanel;
      mark = newMark;
      location = newLocation;
   }

   // property SquarePanel; the panel which the square represents
   public Panel SquarePanel
   {
      get
      {
         return panel;
      }
   } // end property SquarePanel

   // property Mark; the mark of the square
   public char Mark
   {
      get
      {
         return mark;
      }

      set
      {
         mark = value;
      }
   } // end property Mark
```

Fig. 22.7 Class **`Square`**. (Part 1 of 2.)

```

      // property Location; the square's location on the board
      public int Location
      {
         get
         {
            return location;
         }
      } // property Location

   } // end class Square
```

Fig. 22.7 Class `Square`. (Part 2 of 2.)

In this chapter, we discussed how to use C#'s networking technologies by providing both connection-oriented (i.e., streams-based) transmission and connectionless (i.e., packet-based) transmission. We showed how to create a simple server and client via stream sockets, then showed how to create a multithreaded server. In Chapter 23, Data Structures and Collections, we discuss how to store data dynamically and discuss several of the key classes that belong to the C# `System.Collections` namespace.

SUMMARY

- Sockets are the fundamental way to perform network communications in the .NET Framework. The term "socket" refers to the Berkeley Sockets Interface, which was developed in 1978 to facilitate network programming with UNIX and was popularized by C and C++ programmers.
- The two most popular types of sockets are stream sockets and datagram sockets.
- Stream sockets provide a connection-oriented service, meaning that one process establishes a connection to another process, and data can flow between the processes in continuous streams.
- Datagram sockets provide a connectionless service that uses messages to transmit data.
- Connectionless services generally offer greater performance but less reliability than connection-oriented services.
- Transmission Control Protocol (TCP) is the preferred protocol for stream sockets. It is a reliable and relatively fast way to send data through a network.
- The User Datagram Protocol (UDP) is the preferred protocol for datagram sockets. UDP is unreliable. There is no guarantee that packets sent with UDP will arrive in the order in which they were sent or that they will arrive at all.
- The establishment of a simple server with TCP and stream sockets in C# requires five steps. Step 1 is to create a `TcpListener` object. This class represents a TCP stream socket that a server can use to receive connections.
- To receive connections, the `TcpListener` must be listening for them. For the `TcpListener` to listen for client connections, its `Start` method must be called (Step 2).
- `TcpListener` method `AcceptSocket` blocks indefinitely until a connection is established, at which point it returns a `Socket` (Step 3).
- Step 4 is the processing phase, in which the server and the client communicate via methods `Read` and `Write` via a `NetworkStream` object.
- When the client and server have finished communicating, the server closes the connection with the `Close` method on the `Socket` (Step 5). Most servers will then, by means of a control loop, return to the `AcceptSocket` call step to wait for another client's connection.

- A port number is a numeric ID number that a process uses to identify itself at a given network address, also known as an Internet Protocol Address (IP Address).
- An individual process running on a computer is identified by an IP address/port number pair. Hence, no two processes can have the same port number at a given IP address.
- The establishment of a simple client requires four steps. In Step 1, we create a **`TcpClient`** to connect to the server. This connection is established through a call to the **`TcpClient`** method **`Connect`** containing two arguments—the server's IP address and the port number
- In Step 2, the **`TcpClient`** uses method **`GetStream`** to get a **`Stream`** to write to and read from the server.
- Step 3 is the processing phase, in which the client and the server communicate.
- Step 4 has the client close the connection by calling the **`Close`** method on the **`NetworkStream`**.
- **`NetworkStream`** methods **`WriteByte`** and **`Write`** can be used to output individual bytes or sets of bytes to the server, respectively.
- **`NetworkStream`** methods **`ReadByte`** and **`Read`** can be used to read individual bytes or sets of bytes from the server, respectively.
- Class **`UdpClient`** is provided for connectionless transmission of data.
- Class **`UdpClient`** methods **`Send`** and **`Receive`** are used to transmit data.
- Class **`IPEndPoint`** represents an endpoint on a network.
- Class **`IPAddress`** represents an Internet Protocol address.
- Multithreaded servers can manage many simultaneous connections with multiple clients.

TERMINOLOGY

`127.0.0.1`
`AcceptSocket` method of class **`TcpListener`**
Berkeley Sockets Interface
`BinaryReader` class
`BinaryWriter` class
`Bind` method of class **`Socket`**
binding a server to a port
block
block until connection received
client
client/server chat
client/server model
`Close` method of class **`Socket`**
`Close` method of class **`TcpClient`**
collaborative applications
`Connect` method of class **`TcpListener`**
connection
connection attempt
connection between client and server terminates
connection port
connection to a server
connectionless service
connectionless transmission with datagrams
connection-oriented service
connection-oriented, streams-based transmission
datagram
datagram socket
duplicate of datagram
echo a packet back to a client
e-mail
`Exit` method of class **`Environment`**
`ExitCode` property of class **`Environment`**
file processing
`GetStream` method of class **`Socket`**
infinite loop
Internet Protocol Addresses (IP Address)
IP Address
`IPAddress` class
`IPEndPoint` class
LAN
Local Area Network (LAN)
`localhost`
loopback IP address
`Loopback static` member of class **`IPAddress`**
Microsoft Internet Explorer
Netscape Communicator

network address
networking as file I/O
NetworkStream class
OpenRead method of class **WebClient**
OpenWrite method of class **WebClient**
packet
pool of threads
port number
protocol
Read method of class **NetworkStream**
ReadByte method of class **NetworkStream**
reading a file on a Web server
ReadString method of class **BinaryReader**
receive a connection
receive data from a server
Receive method of class **Socket**
Receive method of class **UdpClient**
ReceiveFrom method of class **Socket**
send data to a server
Send method of class **Socket**
Send method of class **UdpClient**
SendTo method of class **Socket**
server
server Internet address
server port number
socket
socket-based communications
Socket class
spawning
Start method of class **TcpListener**
stream
stream socket
streams-based transmission
system service
System.Net namespace
System.Net.Sockets namespace
TcpClient class
TcpListener class
telephone system
Thread class
Transmission Control Protocol (TCP)
UdpClient class
User Datagram Protocol (UDP)
Web server
WebClient class
Write method of class **BinaryWriter**
Write method of class **NetworkStream**
WriteByte method of class **NetworkStream**

SELF-REVIEW EXERCISES

22.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) UDP is a connection-oriented protocol.
b) With stream sockets, a process establishes a connection to another process.
c) Datagram-packet transmission over a network is reliable—packets are guaranteed to arrive in sequence.
d) Most of the time TCP protocol is preferred over the UDP protocol.
e) Each **TcpListener** can accept only one connection.
f) A **TcpListener** can listen for connections at more than one port at a time.
g) A **UdpClient** can send information only to one particular port.
h) Packets sent via a UDP connection are sent only once.
i) Clients need to know the port number at which the server is waiting for connections.

22.2 Fill in the blanks in each of the following statements:

a) Many of C#'s networking classes are contained in namespaces __________ and __________.
b) Class __________ is used for unreliable but fast datagram transmission.
c) An object of class __________ represents an Internet Protocol (IP) address.
d) The two types of sockets we discussed in this chapter are __________ sockets and __________ sockets.
e) The acronym TCP stands for __________.
f) Class __________ listens for connections from clients.
g) Class __________ connects to servers.
h) Class __________ provides access to stream data on a network.

ANSWERS TO SELF-REVIEW EXERCISES

22.1 a) False. UDP is a connectionless protocol, and TCP is a connection-oriented protocol. b) True. c) False. Packets can be lost, arrive out of order or even be duplicated. d) True. e) False. **`TcpListener AcceptSocket`** may be called as often as necessary—each call will accept a new connection. f) False. A **`TcpListener`** can listen for connections at only one port at a time. g) False. A **`UdpClient`** can send information to any port represented by an **`IPEndPoint`**. h) False. Packets may be sent more than once, to make it more likely that at least one copy of each packet arrives. i) True.

22.2 a) **`System.Net`**, **`System.Net.Sockets`**. b) **`UdpClient`**. c) **`IPAddress`**. d) stream, datagram. e) Transmission Control Protocol. f) **`TcpListener`**. g) **`TcpClient`**. h) **`NetworkStream`**.

EXERCISES

22.3 Use a socket connection to allow a client to specify a file name and have the server send the contents of the file or indicate that the file does not exist. Allow the client to modify the file contents and to send the file back to the server for storage.

22.4 Multithreaded servers are quite popular today, especially because of the increasing use of multiprocessing servers (i.e., servers with more than one processor unit). Modify the simple server application presented in Section 22.4 to be a multithreaded server. Then, use several client applications and have each of them connect to the server simultaneously.

22.5 Create a client/server application for the game of Hangman, using socket connections. The server should randomly pick a word or phrase from a file or a database. After connecting, the client should be allowed to begin guessing. If a client guesses incorrectly five times, the game is over. Display the original phrase or word on the server. Display underscores (for letters that have not been guessed yet) and the letters that have been guessed in the word or phrase on the client.

22.6 Modify the previous exercise to be a connectionless game using datagrams.

22.7 *(Modifications to the Multithreaded Tic-Tac-Toe Program)* The programs of Fig. 22.5–Fig. 22.7 implement a multithreaded, client/server version of the game Tic-Tac-Toe. Our goal in developing this game was to demonstrate a multithreaded server that could process multiple connections from clients at the same time. The server in the example is really a mediator between the two clients—it makes sure that each move is valid and that each client moves in the proper order. The server does not determine who won or lost or whether there was a draw. Also, there is no capability to allow a new game to be played or to terminate an existing game.

The following is a list of suggested modifications to the multithreaded Tic-Tac-Toe application:

a) Modify class **`Server`** to test for a win, loss or draw on each move in the game. When the game is over, send a message to each client that indicates the result of the game.

b) Modify class **`Client`** to display a button that, when clicked, allows the client to play another game. The button should be enabled only when a game completes. Note that both class **`Client`** and class **`Server`** must be modified to reset the board and all state information. Also, the other **`Client`** should be notified of a new game, so that client can reset its board and state information.

c) Modify class **`Client`** to provide a button that allows a client to terminate the program at any time. When the button is clicked, the server and the other client should be notified. The server should then wait for a connection from another client so that a new game can begin.

d) Modify class **`Client`** and class **`Server`** so that the loser of a game can choose game piece X or O for the next game. Remember that X always goes first.

22.8 *(Networked Morse Code)* Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 22.8.

Write an application that reads an English-language phrase and encodes the phrase into Morse code. Also, write a program that reads a phrase in Morse code and converts the phrase into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word. Then, enable these two applications to send Morse Code messages to each other through a multithreaded-server application. Each application should allow the user to type normal characters into a `TextBox`. The application should then translate the characters into Morse Code and send the coded message through the server to the other client. When messages are received, they should be decoded and displayed as normal characters and as Morse Code. The application should have two `TextBox`es: One for displaying the other client's messages, and one for typing.

Character	Code	Character	Code
A	•-	T	-
B	-•••	U	••-
C	-•-•	V	•••-
D	-••	W	•--
E	•	X	-••-
F	••-•	Y	-•--
G	--•	Z	--••
H	••••		
I	••	Digits	
J	•---	1	•----
K	-•-	2	••---
L	•-••	3	•••--
M	--	4	••••-
N	-•	5	•••••
O	---	6	-••••
P	•--•	7	--•••
Q	--•-	8	---••
R	•-•	9	----•
S	•••	0	-----

Fig. 22.8 English letters of the alphabet and decimal digits as expressed in international Morse code.

23

Data Structures and Collections

Objectives

- To be able to form linked data structures using references, self-referential classes and recursion.
- To be able to create and manipulate dynamic data structures such as linked lists, queues, stacks and binary trees.
- To understand various important applications of linked data structures.
- To understand how to create reusable data structures with classes, inheritance and composition.

Much that I bound, I could not free;
Much that I freed returned to me.
Lee Wilson Dodd

'Will you walk a little faster?' said a whiting to a snail, 'There's a porpoise close behind us, and he's treading on my tail.'
Lewis Carroll

There is always room at the top.
Daniel Webster

Push on—keep moving.
Thomas Morton

I think that I shall never see
A poem lovely as a tree.
Joyce Kilmer

Outline

23.1 Introduction
23.2 Self-Referential Classes
23.3 Linked Lists
23.4 Stacks
23.5 Queues
23.6 Trees
  23.6.1 Binary Search Tree of Integer Values
  23.6.2 Binary Search Tree of `IComparable` Objects
23.7 Collection Classes
  23.7.1 Class `Array`
  23.7.2 Class `ArrayList`
  23.7.3 Class `Stack`
  23.7.4 Class `Hashtable`
Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

23.1 Introduction

The *data structures* that we have studied thus far have had fixed size, such as single- and double-subscripted arrays. This chapter introduces *dynamic data structures* that grow and shrink at execution time. *Linked lists* are collections of data items “lined up in a row”—users can make insertions and deletions anywhere in a linked list. *Stacks* are important in compilers and operating systems because insertions and deletions are made at only one end—its *top*. *Queues* represent waiting lines; insertions are made at the back (also referred to as the *tail*) of a queue, and deletions are made from the front (also referred to as the *head*) of a queue. *Binary trees* facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file system directories and compilation of expressions into machine language. These data structures have many other interesting applications as well.

We will discuss each of the major types of data structures and implement programs that create and manipulate them. We use classes, inheritance and composition to create and package these data structures for reusability and maintainability.

The chapter examples are practical programs that will be useful in more advanced courses and in industrial applications. The programs devote special attention to and focus on reference manipulation. The exercises offer a rich collection of useful applications.

23.2 Self-Referential Classes

A *self-referential class* contains a reference member that refers to an object of the same class type. For example, the class definition in Fig. 23.1 defines a type, **`Node`**. This type has two **`private`** instance variables—integer **`data`** and **`Node`** reference **`next`**. Member **`next`** references an object of type **`Node`**, an object of the same type as the one being declared here—hence, the term “self-referential class.” Member **`next`** is referred to as a *link* (i.e., **`next`** can be used to “tie” an object of type **`Node`** to another object of the same type). Class **`Node`** also has two properties: One for variable **`data`** (named **`Data`**), and another for variable **`next`** (named **`Next`**).

Self-referential objects can be linked together to form useful data structures, such as lists, queues, stacks and trees. Figure 23.2 illustrates two self-referential objects linked together to form a list. A backslash (representing a **`null`** reference) is placed in the link member of the second self-referential object to indicate that the link does not refer to another object. The slash is for illustration purposes; it does not correspond to the backslash character in C#. A **`null`** reference normally indicates the end of a data structure.

Common Programming Error 23.1

Not setting the link in the last node of a list (or other linear data structure) to **`null`** *is a common logic error.*

```
class Node
{
   private int data;
   private Node next;

   public Node( int d )
   {
      // constructor body
   }

   public int Data
   {
      get
      {
         // get body
      }

      set
      {
         // set body
      }
   }

   public Node Next
   {
      get
      {
         // get body
      }

      set
      {
         // set body
      }
   }
}
```

Fig. 23.1 Sample self-referential **`Node`** class definition.

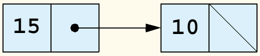


Fig. 23.2 Two self-referential class objects linked together.

Creating and maintaining dynamic data structures requires *dynamic memory allocation*—a program's ability to obtain more memory space at execution time to hold new nodes and to release space no longer needed. As we have already learned, C# programs do not explicitly release dynamically allocated memory. Rather, C# performs automatic garbage collection.

The limit for dynamic memory allocation can be as large as the amount of available disk space in a virtual-memory system. Often, the limits are much smaller, because the computer's available memory must be shared among many users.

Operator **`new`** is essential to dynamic memory allocation. Operator **`new`** takes as an operand the type of the object being dynamically allocated and returns a reference to a newly created object of that type. For example, the statement

```
Node nodeToAdd = new Node( 10 );
```

allocates the appropriate amount of memory to store a **`Node`** and stores a reference to this object in **`nodeToAdd`**. If no memory is available, **`new`** throws an **`OutOfMemoryException`**. The 10 is the **`Node`** object's data.

The following sections discuss lists, stacks, queues and trees. These data structures are created and maintained with dynamic memory allocation and self-referential classes.

Good Programming Practice 23.1

When creating a very large number of objects, test for an **`OutOfMemoryException`**. *Perform appropriate error processing if the requested memory is not allocated.*

23.3 Linked Lists

A *linked list* is a linear collection (i.e., a sequence) of self-referential class objects, called *nodes,* connected by reference links—hence, the term "linked" list. A program accesses a linked list via a reference to the first node of the list. Each subsequent node is accessed via the link-reference member stored in the previous node. By convention, the link reference in the last node of a list is set to **`null`** to mark the end of the list. Data are stored in a linked list dynamically—that is, each node is created as necessary. A node can contain data of any type, including objects of other classes. Stacks and queues are also linear data structures, and they are constrained versions of linked lists. Trees are nonlinear data structures.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is unpredictable. Unlike a linked list, the size of a conventional C# array cannot be altered, because the array size is fixed at creation time. Conventional arrays can become full, but linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Performance Tip 23.1

An array can be declared to contain more elements than the number of items expected, at the expense of wasting memory. Linked lists provide better memory utilization in these situations and they allow the program to adapt at run time.

Performance Tip 23.2

After locating the insertion point for a new item in a sorted linked list, inserting an element in the list is fast—only two references have to be modified. All existing nodes remain at their current locations in memory.

Programmers can maintain linked lists in sorted order simply by inserting each new element at the proper point in the list (locating the proper insertion point does take time). They do not need to move existing list elements.

Performance Tip 23.3

The elements of an array are stored contiguously in memory to allow immediate access to any array element—the address of any element can be calculated directly from its offset from the beginning of the array. Linked lists do not afford such immediate access to their elements—an element can be accessed only by traversing the list from the front.

Memory does not normally store linked list nodes contiguously. Rather, the nodes are logically contiguous. Figure 23.3 illustrates a linked list with several nodes.

Performance Tip 23.4

Using dynamic memory allocation (instead of arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that references occupy space, and that dynamic memory allocation incurs the overhead of method calls.

The program of Fig. 23.4–Fig. 23.5 uses an object of class **`List`** to manipulate a list of miscellaneous object types. The **`Main`** method of class **`ListTest`** (Fig. 23.5) creates a list of objects, inserts objects at the beginning of the list using **`List`** method **`InsertAtFront`**, inserts objects at the end of the list using **`List`** method **`InsertAtBack`**, deletes objects from the front of the list using **`List`** method **`RemoveFromFront`** and deletes objects from the end of the list using **`List`** method **`RemoveFromBack`**. Each insertion and deletion operation invokes **`List`** method **`Print`** to display the current list contents. A detailed discussion of the program follows. If an attempt is made to remove an item from an empty list, an **`EmptyListException`** occurs.

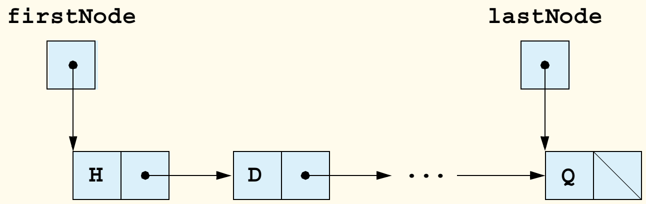


Fig. 23.3 A graphical representation of a linked list.

Performance Tip 23.5

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

The program consists of four classes—**ListNode** (Fig. 23.4, lines 9–52), **List** (Fig. 23.4, lines 55–193), **EmptyListException** (Fig. 23.4, lines 196–203) and class **ListTest** (Fig. 23.5). The classes in Fig. 23.4 create a linked-list library (defined in namespace **LinkedListLibrary**) that can be reused throughout this chapter.

Encapsulated in each **List** object is a linked list of **ListNode** objects. Class **ListNode** (Fig. 23.4, lines 9–52) consists of two member variables—**data** and **next**. Member **data** can refer to any object. Member **next** stores a reference to the next **ListNode** object in the linked list. A **List** accesses the **ListNode** member variables via the properties **Data** (lines 44–50) and **Next** (lines 30–41), respectively.

Class **List** contains **private** members **firstNode** (a reference to the first **ListNode** in a **List**) and **lastNode** (a reference to the last **ListNode** in a **List**). The constructors (lines 62–66 and 69–71) initialize both references to **null**. **InsertAtFront** (lines 76–87), **InsertAtBack** (lines 92–104), **RemoveFromFront** (lines 107–125) and **RemoveFromBack** (lines 128–156) are the primary methods of class **List**. Each method uses a **lock** block to ensure that **List** objects are *multithread safe* when used in a multithreaded program. If one thread is modifying the contents of a **List** object, no other thread can modify the same **List** object at the same time. Method **IsEmpty** (lines 159–165) is a *predicate method* that determines whether the list is empty (i.e., the reference to the first node of the list is **null**). Predicate methods typically test a condition and do not modify the object on which they are called. If the list is empty, method **IsEmpty** returns **true**; otherwise, it returns **false**. Method **Print** (lines 168–191) displays the list's contents. Both **IsEmpty** and **Print** also use **lock** blocks so that the state of the list does not change while those methods are performing their tasks.

Class **EmptyListException** (lines 196–203) defines an exception class to handle illegal operations on an empty **List**.

Class **ListTest** (Fig. 23.5) uses the linked-list library to create and manipulate a linked list. Line 14 creates a new instance of type **List** named **list**. Lines 17–20 create data to add to the list. Lines 23–30 use **List** insertion methods to insert these objects and use **List** method **Print** to output the contents of **list** after each insertion. The code inside the **try** block (lines 36–53) removes objects via **List** deletion methods, outputs the object removed and outputs **list** after every deletion. If there is an attempt to remove an object from an empty list, this **try** block catches the **EmptyListException**. Note that class **ListTest** uses namespace **LinkedListLibrary** (Fig. 23.4); thus, the solution for class **ListTest** must have a reference to the **LinkedListLibrary** class library.

Over the next several pages, we discuss each of the methods of class **List** in detail. Method **InsertAtFront** (Fig. 23.4, lines 76–87) places a new node at the front of the list. The method consists of three steps (illustrated in Fig. 23.6):

1. Call **IsEmpty** to determine whether the list is empty (line 80).

2. If the list is empty, set both **firstNode** and **lastNode** to refer to a new **ListNode** initialized with **insertItem** (lines 81–82). The **ListNode** constructor at lines 16–19 (Fig. 23.4) calls the **ListNode** constructor at lines 23–27

(Fig. 23.4) to set instance variable **`data`** to refer to the **`object`** passed as the first argument and sets the **`next`** reference to **`null`**.

3. If the list is not empty, the new node is “threaded” (not to be confused with *multithreading*) into the list by setting **`firstNode`** to refer to a new **`ListNode`** object initialized with **`insertItem`** and **`firstNode`** (lines 84–85). When the **`ListNode`** constructor (lines 23–27 of Fig. 23.4) executes, it sets instance variable **`data`** to refer to the **`object`** passed as the first argument and performs the insertion by setting the **`next`** reference to the **`ListNode`** passed as the second argument.

```
 1  // Fig. 23.4: LinkedListLibrary.cs
 2  // Class ListNode and class List definitions.
 3
 4  using System;
 5
 6  namespace LinkedListLibrary
 7  {
 8     // class to represent one node in a list
 9     class ListNode
10     {
11        private object data;
12        private ListNode next;
13
14        // constructor to create ListNode that refers to dataValue
15        // and is last node in list
16        public ListNode( object dataValue )
17           : this( dataValue, null )
18        {
19        }
20
21        // constructor to create ListNode that refers to dataValue
22        // and refers to next ListNode in List
23        public ListNode( object dataValue, ListNode nextNode )
24        {
25           data = dataValue;
26           next = nextNode;
27        }
28
29        // property Next
30        public ListNode Next
31        {
32           get
33           {
34              return next;
35           }
36
37           set
38           {
39              next = value;
40           }
41        }
```

Fig. 23.4 Definitions of classes **`ListNode`**, **`List`** and **`EmptyListException`**. (Part 1 of 5.)

```

      // property Data
      public object Data
      {
         get
         {
            return data;
         }
      }

   } // end class ListNode

   // class List definition
   public class List
   {
      private ListNode firstNode;
      private ListNode lastNode;
      private string name;     // string like "list" to display

      // construct empty List with specified name
      public List( string listName )
      {
         name = listName;
         firstNode = lastNode = null;
      }

      // construct empty List with "list" as its name
      public List() : this( "list" )
      {
      }

      // Insert object at front of List. If List is empty,
      // firstNode and lastNode will refer to same object.
      // Otherwise, firstNode refers to new node.
      public void InsertAtFront( object insertItem )
      {
         lock ( this )
         {
            if ( IsEmpty() )
               firstNode = lastNode =
                  new ListNode( insertItem );
            else
               firstNode =
                  new ListNode( insertItem, firstNode );
         }
      }

      // Insert object at end of List. If List is empty,
      // firstNode and lastNode will refer to same object.
      // Otherwise, lastNode's Next property refers to new node.
      public void InsertAtBack( object insertItem )
      {
```

Fig. 23.4 Definitions of classes **`ListNode`**, **`List`** and **`EmptyListException`**. (Part 2 of 5.)

```
         lock ( this )
         {
            if ( IsEmpty() )
               firstNode = lastNode =
                  new ListNode( insertItem );

            else
               lastNode = lastNode.Next =
                  new ListNode( insertItem );
         }
      }

      // remove first node from List
      public object RemoveFromFront()
      {
         lock ( this )
         {
            if ( IsEmpty() )
               throw new EmptyListException( name );

            object removeItem = firstNode.Data;   // retrieve data

            // reset firstNode and lastNode references
            if ( firstNode == lastNode )
               firstNode = lastNode = null;

            else
               firstNode = firstNode.Next;

            return removeItem;  // return removed data
         }
      }

      // remove last node from List
      public object RemoveFromBack()
      {
         lock ( this )
         {
            if ( IsEmpty() )
               throw new EmptyListException( name );

            object removeItem = lastNode.Data;   // retrieve data

            // reset firstNode and lastNode references
            if ( firstNode == lastNode )
               firstNode = lastNode = null;

            else
            {
               ListNode current = firstNode;

```

Fig. 23.4 Definitions of classes `ListNode`, `List` and `EmptyListException`. (Part 3 of 5.)

```
               // loop while current node is not lastNode
               while ( current.Next != lastNode )
                  current = current.Next;    // move to next node

               // current is new lastNode
               lastNode = current;
               current.Next = null;
            }

            return removeItem;  // return removed data
         }
      }

      // return true if List is empty
      public bool IsEmpty()
      {
         lock ( this )
         {
            return firstNode == null;
         }
      }

      // output List contents
      virtual public void Print()
      {
         lock ( this )
         {
            if ( IsEmpty() )
            {
               Console.WriteLine( "Empty " + name );
               return;
            }

            Console.Write( "The " + name + " is: " );

            ListNode current = firstNode;

            // output current node data while not at end of list
            while ( current != null )
            {
               Console.Write( current.Data + " " );
               current = current.Next;
            }

            Console.WriteLine( "\n" );
         }
      }

   } // end class List

```

Fig. 23.4 Definitions of classes **ListNode**, **List** and **EmptyListException**. (Part 4 of 5.)

```
   // class EmptyListException definition
   public class EmptyListException : ApplicationException
   {
      public EmptyListException( string name )
         : base( "The " + name + " is empty" )
      {
      }

   } // end class EmptyListException

} // end namespace LinkedListLibrary
```

Fig. 23.4 Definitions of classes `ListNode`, `List` and `EmptyListException`. (Part 5 of 5.)

```
// Fig 23.5: ListTest.cs
// Testing class List.

using System;
using LinkedListLibrary;

namespace ListTest
{
   // class to test List class functionality
   class ListTest
   {
      static void Main( string[] args )
      {
         List list = new List();  // create List container

         // create data to store in List
         bool aBoolean = true;
         char aCharacter = '$';
         int anInteger =  34567;
         string aString = "hello";

         // use List insert methods
         list.InsertAtFront( aBoolean );
         list.Print();
         list.InsertAtFront( aCharacter );
         list.Print();
         list.InsertAtBack( anInteger );
         list.Print();
         list.InsertAtBack( aString );
         list.Print();

         // use List remove methods
         object removedObject;

```

Fig. 23.5 Demonstrating the linked list. (Part 1 of 2.)

```
         // remove data from list and print after each removal
         try
         {
            removedObject = list.RemoveFromFront();
            Console.WriteLine( removedObject + " removed" );
            list.Print();

            removedObject = list.RemoveFromFront();
            Console.WriteLine( removedObject + " removed" );
            list.Print();

            removedObject = list.RemoveFromBack();
            Console.WriteLine( removedObject + " removed" );
            list.Print();

            removedObject = list.RemoveFromBack();
            Console.WriteLine( removedObject + " removed" );
            list.Print();
         }

         // process exception if list empty when attempt is
         // made to remove item
         catch ( EmptyListException emptyListException )
         {
            Console.Error.WriteLine( "\n" + emptyListException );
         }

      } // end method Main

   } // end class ListTest
}
```

```
The list is: True

The list is: $ True

The list is: $ True 34567

The list is: $ True 34567 hello

$ removed
The list is: True 34567 hello

True removed
The list is: 34567 hello

hello removed
The list is: 34567

34567 removed
Empty list
```

Fig. 23.5 Demonstrating the linked list. (Part 2 of 2.)

Fig. 23.6 illustrates method **`InsertAtFront`**. Part (a) of the figure shows the list and the new node during the **`InsertAtFront`** operation and before the threading of the new node into the list. The dotted arrows in part (b) illustrate step 3 of the **`InsertAtFront`** operation, which enables the node containing **`12`** to become the new list front.

Method **`InsertAtBack`** (Fig. 23.4, lines 92–104) places a new node at the back of the list. The method consists of three steps (illustrated in Fig. 23.7):

1. Call **`IsEmpty`** to determine whether the list is empty (line 96).
2. If the list is empty, set both **`firstNode`** and **`lastNode`** to refer to a new **`ListNode`** initialized with **`insertItem`** (lines 97–98). The **`ListNode`** constructor at lines 16–19 (Fig. 23.4) calls the **`ListNode`** constructor at lines 23–27 (Fig. 23.4) to set instance variable **`data`** to refer to the **`object`** passed as the first argument and sets the **`next`** reference to **`null`**.
3. If the list is not empty, thread the new node into the list by setting **`lastNode`** and **`lastNode.next`** to refer to a new **`ListNode`** object initialized with **`insertItem`** (lines 101–102). When the **`ListNode`** constructor (lines 16–19 of Fig. 23.4) executes, it sets instance variable **`data`** to refer to the **`object`** passed as an argument and sets the **`next`** reference to **`null`**.

Fig. 23.7 illustrates an **`InsertAtBack`** operation. Part a) of the figure shows the list and the new node during the **`InsertAtBack`** operation and before the new node has been threaded into the list. The dotted arrows in part b) illustrate the steps of method **`InsertAtBack`** that enable a new node to be added to the end of a list that is not empty.

Method **`RemoveFromFront`** (Fig. 23.4, lines 107–127) removes the front node of the list and returns a reference to the removed data. The method throws an **`EmptyListException`** (line 114) if the programmer tries to remove a node from an empty list. Otherwise, the method returns a reference to the removed data. The method consists of four steps (illustrated in Fig. 23.8):

1. Assign **`firstNode.Data`** (the data being removed from the list) to reference **`removeItem`** (line 116).
2. If the objects to which **`firstNode`** and **`lastNode`** refer are the same object, the list has only one element prior to the removal attempt. In this case, the method sets **`firstNode`** and **`lastNode`** to **`null`** (line 120) to “dethread” (remove) the node from the list (leaving the list empty).
3. If the list has more than one node prior to removal, then the method leaves reference **`lastNode`** as is and simply assigns **`firstNode.Next`** to reference **`firstNode`** (line 123). Thus, **`firstNode`** references the node that was the second node prior to the **`RemoveFromFront`** call.
4. Return the **`removeItem`** reference.

Fig. 23.8 illustrates method **`RemoveFromFront`**. Part a) illustrates the list before the removal operation. Part b) shows actual reference manipulations.

Method **`RemoveFromBack`** (Fig. 23.4, lines 130–160) removes the last node of a list and returns a reference to the removed data. The method throws an **`EmptyListException`** (line 137) if the program attempts to remove a node from an empty list. The method consists of several steps (illustrated in Fig. 23.9):

1. Assign `lastNode.Data` (the data being removed from the list) to reference `removeItem` (line 139).
2. If the objects to which `firstNode` and `lastNode` refer are the same object (line 142), the list has only one element prior to the removal attempt. In this case, the method sets `firstNode` and `lastNode` to `null` (line 143) to dethread (remove) that node from the list (leaving the list empty).

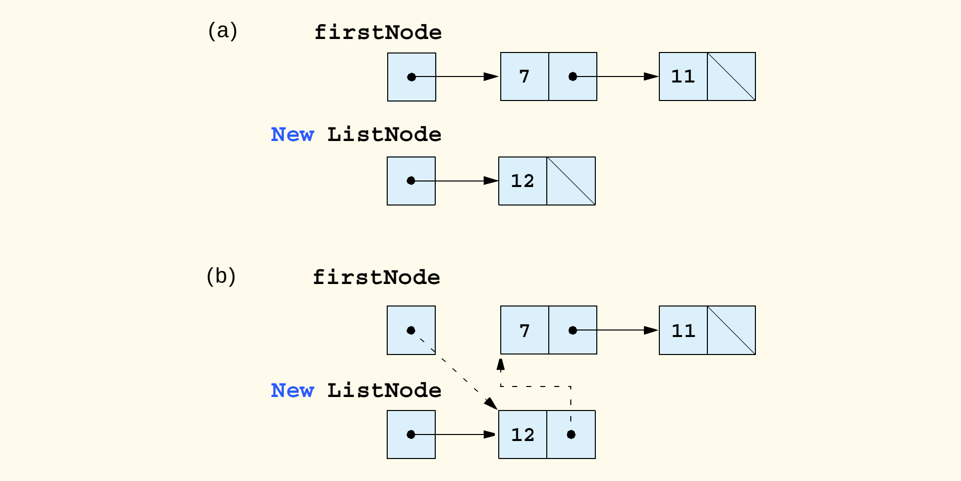

(a)
firstNode
7
11
New ListNode
12
(b)
firstNode
7
11
New ListNode
12

Fig. 23.6 A graphical representation of the `InsertAtFront` operation.

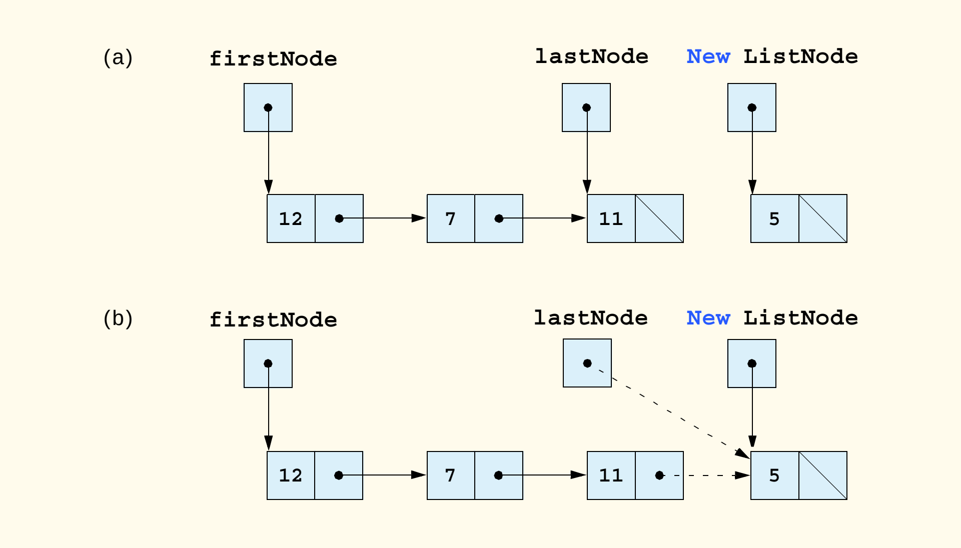

(a)
firstNode
lastNode
New ListNode
12
7
11
5
(b)
firstNode
lastNode
New ListNode
12
7
11
5

Fig. 23.7 A graphical representation of the `InsertAtBack` operation.

3. If the list has more than one node prior to removal, create the **`ListNode`** reference **`current`** and assign it **`firstNode`** (line 147).
4. Now "walk the list" with **`current`** until it references the node before the last node. The **`while`** loop (lines 150–151) assigns **`current.Next`** to reference **`current`** as long as **`current.Next`** is not equal to **`lastNode`**.
5. After locating the second-to-last node, assign **`current`** to **`lastNode`** (line 154) to dethread the last node from the list.
6. Set **`current.Next`** to **`null`** (line 155) in the new last node of the list to ensure proper list termination.
7. Return the **`removeItem`** reference (line 140).

Fig. 23.9 illustrates method **`RemoveFromBack`**. Part a) illustrates the list before the removal operation. Part b) shows the actual reference manipulations.

Method **`Print`** (Fig. 23.4, lines 172–195) first determines whether the list is empty (line 176). If so, **`Print`** displays a **`string`** consisting of the string **`"Empty "`** and the list's **`name`**, then returns control to the calling method. Otherwise, **`Print`** outputs the data in the list. The method prints a string consisting of the string **`"The "`**, the **`name`** and the string **`" is: "`**. Then, line 184 creates **`ListNode`** reference **`current`** and initializes it with **`firstNode`**. While **`current`** is not **`null`**, there are more items in the list. Therefore, the method prints **`current.Data`** (line 189), then assigns **`current.Next`** to **`current`** (line 190) to move to the next node in the list. Note that, if the link in the last node of the list is not **`null`**, the printing algorithm will erroneously attempt to print past the end of the list. The printing algorithm is identical for linked lists, stacks and queues.

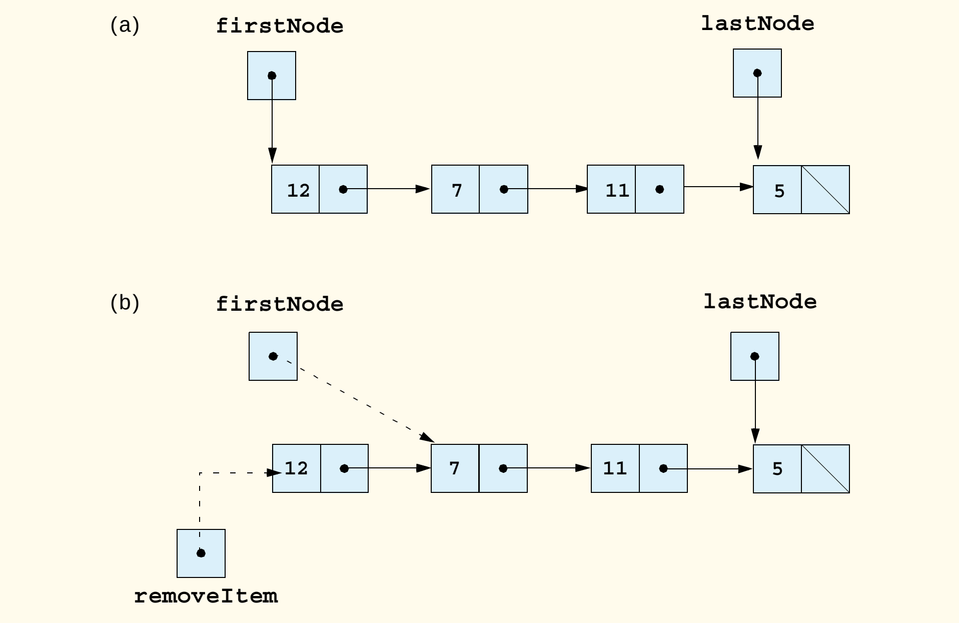


Fig. 23.8 A graphical representation of the **`RemoveFromFront`** operation.

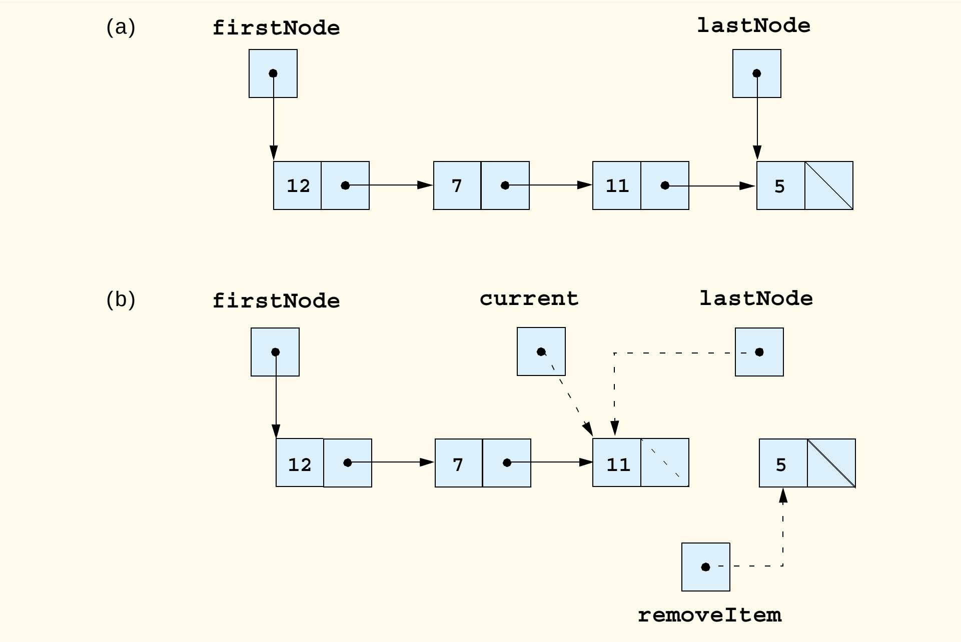


Fig. 23.9 A graphical representation of the **`RemoveFromBack`** operation.

23.4 Stacks

A *stack* is a constrained version of a linked list—a stack takes new nodes and releases nodes only at the top. For this reason, a stack is referred to as a *last-in, first-out* (*LIFO*) data structure. The link member in the bottom (i.e., last) node of the stack is set to **`null`** to indicate the bottom of the stack.

The primary operations to manipulate a stack are *push* and *pop*. Operation push adds a new node to the top of the stack. Operation pop removes a node from the top of the stack and returns the item from the popped node.

Stacks have many interesting applications. For example, when a program calls a method, the called method must know how to return to its caller, so the return address is pushed onto the *program execution stack*. If a series of method calls occurs, the successive return values are pushed onto the stack in last-in, first-out order so that each method can return to its caller. Stacks support recursive method calls in the same manner that they do conventional nonrecursive method calls.

The program-execution stack contains the space created for local variables on each invocation of a method during a program's execution. When the method returns to its caller, the space for that method's local variables is popped off the stack, and those variables are no longer known to the program.

The ***`System.Collections`*** namespace contains class **`Stack`** for implementing and manipulating stacks that can grow and shrink during program execution. Section 23.7 discusses class **`Stack`**.

We take advantage of the close relationship between lists and stacks to implement a stack class by reusing a list class. We demonstrate two different forms of reusability. First, we implement the stack class by inheriting from class `List` of Fig. 23.4. Then, we implement an identically performing stack class through composition by including a `List` object as a `private` member of a stack class. This chapter implements list, stack and queue data structures to store `object` references to encourage further reusability. Thus, any object type can be stored in a list, stack or queue.

The program of Fig. 23.10 and Fig. 23.11 creates a stack class by inheriting from class `List` of Fig. 23.4. We want the stack to have methods `Push`, `Pop`, `IsEmpty` and `Print`. Essentially, these are the methods `InsertAtFront`, `RemoveFromFront`, `IsEmpty` and `Print` of class `List`. Of course, class `List` contains other methods (such as `InsertAtBack` and `RemoveFromBack`) that we would rather not make accessible through the `public` interface of the stack. It is important to remember that all methods in the `public` interface of class `List` are also `public` methods of the derived class `StackInheritance` (Fig. 23.10).

When we implement the stack's methods, we have each `StackInheritance` method call the appropriate `List` method—method `Push` calls `InsertAtFront`, method `Pop` calls `RemoveFromFront`. Class `StackInheritance` does not define methods `IsEmpty` and `Print`, because `StackInheritance` inherits these methods from class `List` into `StackInheritance`'s `public` interface. The methods in class `StackInheritance` do not use `lock` statements. Each of the methods in this class calls a method from class `List` that uses `lock`. If two threads call `Push` on the same stack object, only one of the threads at a time will be able to call `List` method `InsertAtFront`. Note that class `StackInheritance` uses namespace `LinkedListLibrary` (Fig. 23.4); thus, the solution for the class library that defines `StackInheritance` must have a reference to the `LinkedListLibrary` class library.

`StackInheritanceTest`'s `Main` method (Fig. 23.11) uses class `StackInheritance` to instantiate a stack of `object`s called `stack`. Lines 18–21 define four objects that will be pushed onto the stack and popped off the stack. The program pushes onto the stack (lines 24, 26, 28 and 30) a `bool` containing `true`, a `char` containing `$`, an `int` containing `34567` and a `string` containing `hello`. An infinite `while` loop (lines 36–41) pops the elements from the stack. When there are no objects left to pop, method `Pop` throws an `EmptyListException` and the program displays the exception's stack trace, which shows the program execution stack at the time the exception occurred. The program uses method `Print` (inherited from class `List`) to output the contents of the stack after each operation. Note that class `StackInheritanceTest` uses namespace `LinkedListLibrary` (Fig. 23.4) and namespace `StackInheritanceLibrary` (Fig. 23.10); thus, the solution for class `StackInheritanceTest` must have references to both class libraries.

```
// Fig. 23.10: StackInheritanceLibrary.cs
// Implementing a stack by inheriting from class List.

using System;
using LinkedListLibrary;
```

Fig. 23.10 `StackInheritance` extends class `List`. (Part 1 of 2.)

```

namespace StackInheritanceLibrary
{
   // class StackInheritance inherits class List's capabilities
   public class StackInheritance : List
   {
      // pass name "stack" to List constructor
      public StackInheritance() : base( "stack" )
      {
      }

      // place dataValue at top of stack by inserting
      // dataValue at front of linked list
      public void Push( object dataValue )
      {
         InsertAtFront( dataValue );
      }

      // remove item from top of stack by removing
      // item at front of linked list
      public object Pop()
      {
         return RemoveFromFront();
      }

   } // end class StackInheritance
}
```

Fig. 23.10 `StackInheritance` extends class `List`. (Part 2 of 2.)

Another way to implement a stack class is by reusing a list class through composition. The class in Fig. 23.12 uses a **`private`** object of class **`List`** (line 12) in the definition of class **`StackComposition`**. Composition enables us to hide the methods of class **`List`** that should not be in our stack's **`public`** interface by providing **`public`** interface methods only to the required **`List`** methods. This class implements each stack method by delegating its work to an appropriate **`List`** method. In particular, **`StackComposition`** calls **`List`** methods **`InsertAtFront`**, **`RemoveFromFront`**, **`IsEmpty`** and **`Print`**. In this example, we do not show class **`StackCompositionTest`**, because the only difference in this example is that we change the type of the stack from **`StackInheritance`** to **`StackComposition`**. If you execute the application from the code on the CD that accompanies this book, you will see that the output is identical.

```
// Fig. 23.11: StackInheritanceTest.cs
// Testing class StackInheritance.

using System;
using StackInheritanceLibrary;
using LinkedListLibrary;

```

Fig. 23.11 Using class `StackInheritance`. (Part 1 of 3.)

```
namespace StackInheritanceTest
{
   // demonstrate functionality of class StackInheritance
   class StackInheritanceTest
   {
      static void Main( string[] args )
      {
         StackInheritance stack = new StackInheritance();

         // create objects to store in the stack
         bool aBoolean = true;
         char aCharacter = '$';
         int anInteger = 34567;
         string aString = "hello";

         // use method Push to add items to stack
         stack.Push( aBoolean );
         stack.Print();
         stack.Push( aCharacter );
         stack.Print();
         stack.Push( anInteger );
         stack.Print();
         stack.Push( aString );
         stack.Print();

         // use method Pop to remove items from stack
         try
         {
            while ( true )
            {
               object removedObject = stack.Pop();
               Console.WriteLine( removedObject + " popped" );
               stack.Print();
            }
         }

         // if exception occurs, print stack trace
         catch ( EmptyListException emptyListException )
         {
            Console.Error.WriteLine(
               emptyListException.StackTrace );
         }

      } // end method Main

   } // end class StackInheritanceTest
}
```

```
The stack is: True

The stack is: $ True
```

(continued on next page)

Fig. 23.11 Using class `StackInheritance`. (Part 2 of 3.)

(continued from previous page)

```
The stack is: 34567 $ True

The stack is: hello 34567 $ True

hello popped
The stack is: 34567 $ True

34567 popped
The stack is: $ True

$ popped
The stack is: True

True popped
Empty stack
   at LinkedListLibrary.List.RemoveFromFront()
      in z:\ch23\linkedlistlibrary\linkedlistlibrary.cs:line 114
   at StackInheritanceLibrary.StackInheritance.Pop()
      in z:\ch23\stackinheritancelibrary\
      stackinheritancelibrary.cs:line 28
   at StackInheritanceTest.StackInheritanceTest.Main(String[] args
      in z:\ch23\fig23_11\stackinheritancetest.cs:line 41
```

Fig. 23.11 Using class `StackInheritance`. (Part 3 of 3.)

```
// Fig. 23.12: StackCompositionLibrary.cs
// StackComposition definition with composed List object.

using System;
using LinkedListLibrary;

namespace StackCompositionLibrary
{
   // class StackComposition encapsulates List's capabilities
   public class StackComposition
   {
      private List stack;

      // construct empty stack
      public StackComposition()
      {
         stack = new List( "stack" );
      }

      // add object to stack
      public void Push( object dataValue )
      {
         stack.InsertAtFront( dataValue );
      }

```

Fig. 23.12 `StackComposition` class encapsulates functionality of class `List`. (Part 1 of 2.)

```
         // remove object from stack
         public object Pop()
         {
            return stack.RemoveFromFront();
         }

         // determine whether stack is empty
         public bool IsEmpty()
         {
            return stack.IsEmpty();
         }

         // output stack contents
         public void Print()
         {
            stack.Print();
         }

      } // end class StackComposition
   }
```

Fig. 23.12 `StackComposition` class encapsulates functionality of class `List`. (Part 2 of 2.)

23.5 Queues

Another common data structure is the *queue***.** A queue is similar to a checkout line in a supermarket—the first person in line is served first; customers enter the line only at the end, and they wait to be served. Queue nodes are removed only from the *head* of the queue and are inserted only at the *tail* of the queue. For this reason, a queue is a *first-in*, *first-out* (*FIFO*) data structure. The insert and remove operations are known as *enqueue* and *dequeue*.

Queues have many applications in computer systems. Most computers have only a single processor, so they can only serve one user at a time. Entries for the other users are placed in a queue. The entry at the front of the queue receives the first available service. Each entry gradually advances to the front of the queue as users receive service.

Queues also support print spooling. A multiuser environment may have only one printer. Several users may send output to the printer. If the printer is busy, users may still generate other outputs, which are "spooled" to disk (much as thread is wound onto a spool), where they wait in a queue until the printer becomes available.

Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, the routing node must route it to the next node on the network along the path to the packet's final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

A file server in a computer network handles file access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When client requests exceed that capacity, the requests wait in queues.

The program of Fig. 23.13 and Fig. 23.14 creates a queue class through inheritance from a list class. We want the `QueueInheritance` class (Fig. 23.13) to have methods `Enqueue`, `Dequeue`, `IsEmpty` and `Print`. Note that these methods are essentially the `InsertAtBack`, `RemoveFromFront`, `IsEmpty` and `Print` methods of class `List`.

Of course, the list class contains other methods (such as `InsertAtFront` and `RemoveFromBack`) that we would rather not make accessible through the `public` interface to the queue class. Remember that all methods in the `public` interface of the `List` class are also `public` methods of the derived class `QueueInheritance`.

When we implement the queue's methods, we have each `QueueInheritance` method call the appropriate `List` method—method `Enqueue` calls `InsertAtBack`, method `Dequeue` calls `RemoveFromFront`, and `IsEmpty` and `Print` calls invoke their base-class versions. Class `QueueInheritance` does not define methods `IsEmpty` and `Print`, because `QueueInheritance` inherits these methods from class `List` into `QueueInheritance`'s `public` interface. Also, the methods in class `QueueInheritance` do not use `lock` statements. Each of the methods in this class calls a method from class `List` that uses `lock`. Note that class `QueueInheritance` uses namespace `LinkedListLibrary` (Fig. 23.4); thus, the solution for the class library that defines `QueueInheritance` must have a reference to the `LinkedListLibrary` class library.

Class `QueueInheritanceTest`'s `Main` method (Fig. 23.14) uses class `QueueInheritance` to instantiate a queue of `object`s called `queue`. Lines 18–21 define four objects that will be enqueued and dequeued. The program enqueues (lines 24, 26, 28 and 30) a `bool` containing `true`, a `char` containing `$`, an `int` containing `34567` and a `string` containing `hello`.

```
// Fig. 23.13: QueueInheritanceLibrary.cs
// Implementing a queue by inheriting from class List.

using System;
using LinkedListLibrary;

namespace QueueInheritanceLibrary
{
   // class QueueInheritance inherits List's capabilities
   public class QueueInheritance : List
   {
      // pass name "queue" to List constructor
      public QueueInheritance() : base( "queue" )
      {
      }

      // place dataValue at end of queue by inserting
      // dataValue at end of linked list
      public void Enqueue( object dataValue )
      {
         InsertAtBack( dataValue );
      }

      // remove item from front of queue by removing
      // item at front of linked list
      public object Dequeue( )
      {
         return RemoveFromFront();
      }
```

Fig. 23.13 `QueueInheritance` extends class `List`. (Part 1 of 2.)

```

   } // end of QueueInheritance
}
```

Fig. 23.13 **`QueueInheritance`** extends class **`List`**. (Part 2 of 2.)

An infinite **`while`** loop (lines 39–44) dequeues the elements from the queue in FIFO order. When there are no objects left to dequeue, method **`Dequeue`** throws an **`EmptyListException`** and the program displays the exception's stack trace, which shows the program execution stack at the time the exception occurred. The program uses method **`Print`** (inherited from class **`List`**) to output the contents of the queue after each operation. Note that class **`QueueInheritanceTest`** uses namespace **`LinkedListLibrary`** (Fig. 23.4) and namespace **`QueueInheritanceLibrary`** (Fig. 23.13); thus, the solution for class **`QueueInheritanceTest`** must have references to both class libraries.

```
1  // Fig. 23.14: QueueTest.cs
2  // Testing class QueueInheritance.
3
4  using System;
5  using QueueInheritanceLibrary;
6  using LinkedListLibrary;
7
8  namespace QueueTest
9  {
10    // demonstrate functionality of class QueueInheritance
11    class QueueTest
12    {
13       static void Main( string[] args )
14       {
15          QueueInheritance queue = new QueueInheritance();
16
17          // create objects to store in the stack
18          bool aBoolean = true;
19          char aCharacter = '$';
20          int anInteger = 34567;
21          string aString = "hello";
22
23          // use method Enqueue to add items to queue
24          queue.Enqueue( aBoolean );
25          queue.Print();
26          queue.Enqueue( aCharacter );
27          queue.Print();
28          queue.Enqueue( anInteger );
29          queue.Print();
30          queue.Enqueue( aString );
31          queue.Print();
32
33          // use method Dequeue to remove items from queue
34          object removedObject = null;
35
```

Fig. 23.14 Using inheritance to create a queue. (Part 1 of 2.)

```
         // remove items from queue
         try
         {
            while ( true )
            {
               removedObject = queue.Dequeue();
               Console.WriteLine( removedObject + " dequeue" );
               queue.Print();
            }
         }

         // if exception occurs, print stack trace
         catch ( EmptyListException emptyListException )
         {
            Console.Error.WriteLine(
               emptyListException.StackTrace );
         }

      } // end method Main

   } // end class QueueTest
}
```

```
The queue is: True

The queue is: True $

The queue is: True $ 34567

The queue is: True $ 34567 hello

True dequeue
The queue is: $ 34567 hello

$ dequeue
The queue is: 34567 hello

34567 dequeue
The queue is: hello

hello dequeue
Empty queue
   at LinkedListLibrary.List.RemoveFromFront()
      in z:\ch23\linkedlistlibrary\linkedlistlibrary.cs:line 114
   at QueueInheritanceLibrary.QueueInheritance.Dequeue()
      in z:\ch23\queueinheritancelibrary\
      queueinheritancelibrary.cs:line 28
   at QueueTest.QueueTest.Main(String[] args)
      in z:\ch23\fig23_14\queuetest.cs:line 41
```

Fig. 23.14 Using inheritance to create a queue. (Part 2 of 2.)

23.6 Trees

Linked lists, stacks and queues are *linear data structures* (i.e., *sequences*). A *tree* is a non-linear, two-dimensional data structure with special properties. Tree nodes contain two or more links. This section discusses *binary trees* (Fig. 23.15)—trees whose nodes all contain two links (none, one or both of which may be **`null`**). The *root node* is the first node in a tree. Each link in the root node refers to a *child*. The *left child* is the first node in the *left subtree*, and the *right child* is the first node in the *right subtree*. The children of a specific node are called *siblings*. A node with no children is called a *leaf node*. Computer scientists normally draw trees from the root node down—exactly the opposite of the way most trees grow in nature.

Common Programming Error 23.2

Not setting to **`null`** *the links in leaf nodes of a tree is a common logic error.*

In our binary tree example, we create a special binary tree called a *binary search tree*. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in the subtree's parent node, and the values in any right subtree are greater than the value in the subtree's parent node. Figure 23.16 illustrates a binary search tree with 12 integer values. Note that the shape of the binary search tree that corresponds to a set of data can depend on the order in which the values are inserted into the tree.

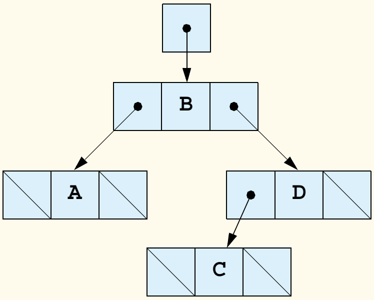


Fig. 23.15 A graphical representation of a binary tree.

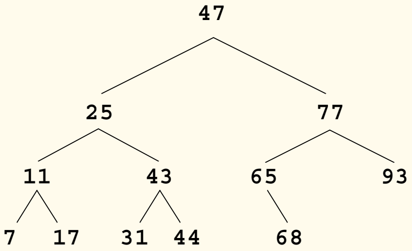


Fig. 23.16 A binary search tree containing 12 values.

23.6.1 Binary Search Tree of Integer Values

The application of Fig. 23.17 and Fig. 23.18 creates a binary search tree of integers and traverses it (i.e., walks through all its nodes) in three ways—using recursive *inorder, preorder* and *postorder traversals*. The program generates 10 random numbers and inserts each into the tree. Figure 23.17 defines class `Tree` in namespace `BinaryTreeLibrary` for reuse purposes. Figure 23.18 defines class `TreeTest` to demonstrate class `Tree`'s functionality. Method `Main` of class `TreeTest` instantiates an empty `Tree` object, then randomly generates 10 integers and inserts each value in the binary tree by calling `Tree` method `InsertNode`. The program then performs preorder, inorder and postorder traversals of the tree. We will discuss these traversals shortly.

Class `TreeNode` (lines 9–95 of Fig. 23.17) is a self-referential class containing three `private` data members—`leftNode` and `rightNode`, of type `TreeNode`, and `data`, of type `int`. Initially, every `TreeNode` is a leaf node, so the constructor (lines 16–20) initializes references `leftNode` and `rightNode` to `null`. Properties `LeftNode` (lines 23–34), `Data` (lines 37–48) and `RightNode` (lines 51–62) provide access to a `ListNode`'s `private` data members. We discuss `TreeNode` method `Insert` (lines 67–93) shortly.

```
 1  // Fig. 23.17: BinaryTreeLibrary.cs
 2  // Definition of class TreeNode and class Tree.
 3
 4  using System;
 5
 6  namespace BinaryTreeLibrary
 7  {
 8     // class TreeNode definition
 9     class TreeNode
10     {
11        private TreeNode leftNode;
12        private int data;
13        private TreeNode rightNode;
14
15        // initialize data and make this a leaf node
16        public TreeNode( int nodeData )
17        {
18           data = nodeData;
19           leftNode = rightNode = null;  // node has no children
20        }
21
22        // LeftNode property
23        public TreeNode LeftNode
24        {
25           get
26           {
27              return leftNode;
28           }
29
```

Fig. 23.17 Definitions of `TreeNode` and `Tree` for a binary search tree. (Part 1 of 5.)

```
            set
            {
               leftNode = value;
            }
         }

         // Data property
         public int Data
         {
            get
            {
               return data;
            }

            set
            {
               data = value;
            }
         }

         // RightNode property
         public TreeNode RightNode
         {
            get
            {
               return rightNode;
            }

            set
            {
               rightNode = value;
            }
         }


         // insert TreeNode into Tree that contains nodes;
         // ignore duplicate values
         public void Insert( int insertValue )
         {
            // insert in left subtree
            if ( insertValue < data )
            {
               // insert new TreeNode
               if ( leftNode == null )
                  leftNode = new TreeNode( insertValue );

               // continue traversing left subtree
               else
                  leftNode.Insert( insertValue );
            }

```

Fig. 23.17 Definitions of `TreeNode` and `Tree` for a binary search tree. (Part 2 of 5.)

```
            // insert in right subtree
            else if ( insertValue > data )
            {
               // insert new TreeNode
               if ( rightNode == null )
                  rightNode = new TreeNode( insertValue );

               // continue traversing right subtree
               else
                  rightNode.Insert( insertValue );
            }

         }  // end method Insert

      }  // end class TreeNode

      // class Tree definition
      public class Tree
      {
         private TreeNode root;

         // construct an empty Tree of integers
         public Tree()
         {
            root = null;
         }

         // Insert a new node in the binary search tree.
         // If the root node is null, create the root node here.
         // Otherwise, call the insert method of class TreeNode.
         public void InsertNode( int insertValue )
         {
            lock ( this )
            {
               if ( root == null )
                  root = new TreeNode( insertValue );

               else
                  root.Insert( insertValue );
            }
         }

         // begin preorder traversal
         public void PreorderTraversal()
         {
            lock ( this )
            {
               PreorderHelper( root );
            }
         }

```

Fig. 23.17 Definitions of `TreeNode` and `Tree` for a binary search tree. (Part 3 of 5.)

```
      // recursive method to perform preorder traversal
      private void PreorderHelper( TreeNode node )
      {
         if ( node == null )
            return;

         // output node data
         Console.Write( node.Data + " " );

         // traverse left subtree
         PreorderHelper( node.LeftNode );

         // traverse right subtree
         PreorderHelper( node.RightNode );
      }

      // begin inorder traversal
      public void InorderTraversal()
      {
         lock ( this )
         {
            InorderHelper( root );
         }
      }

      // recursive method to perform inorder traversal
      private void InorderHelper( TreeNode node )
      {
         if ( node == null )
            return;

         // traverse left subtree
         InorderHelper( node.LeftNode );

         // output node data
         Console.Write( node.Data + " " );

         // traverse right subtree
         InorderHelper( node.RightNode );
      }

      // begin postorder traversal
      public void PostorderTraversal()
      {
         lock ( this )
         {
            PostorderHelper( root );
         }
      }

```

Fig. 23.17 Definitions of `TreeNode` and `Tree` for a binary search tree. (Part 4 of 5.)

```
      // recursive method to perform postorder traversal
      private void PostorderHelper( TreeNode node )
      {
         if ( node == null )
            return;

         // traverse left subtree
         PostorderHelper( node.LeftNode );

         // traverse right subtree
         PostorderHelper( node.RightNode );

         // output node data
         Console.Write( node.Data + " " );
      }

   }  // end class Tree
}
```

Fig. 23.17 Definitions of **`TreeNode`** and **`Tree`** for a binary search tree. (Part 5 of 5.)

```
// Fig. 23.18: TreeTest.cs
// This program tests class Tree.

using System;
using BinaryTreeLibrary;

namespace TreeTest
{
   // class TreeTest definition
   public class TreeTest
   {
      // test class Tree
      static void Main( string[] args )
      {
         Tree tree = new Tree();
         int insertValue;

         Console.WriteLine( "Inserting values: " );
         Random random = new Random();

         // insert 10 random integers from 0-99 in tree
         for ( int i = 1; i <= 10; i++ )
         {
            insertValue = random.Next( 100 );
            Console.Write( insertValue + " " );

            tree.InsertNode( insertValue );
         }

         // perform preorder traversal of tree
         Console.WriteLine( "\n\nPreorder traversal" );
```

Fig. 23.18 Creating and traversing a binary tree. (Part 1 of 2.)

```
         tree.PreorderTraversal();

         // perform inorder traversal of tree
         Console.WriteLine( "\n\nInorder traversal" );
         tree.InorderTraversal();

         // perform postorder traversal of tree
         Console.WriteLine( "\n\nPostorder traversal" );
         tree.PostorderTraversal();
         Console.WriteLine();
      }

   }  // end class TreeTest
}
```

```
Inserting values:
39 69 94 47 50 72 55 41 97 73

Preorder traversal
39 69 47 41 50 55 94 72 73 97

Inorder traversal
39 41 47 50 55 69 72 73 94 97

Postorder traversal
41 55 50 47 73 72 97 94 69 39
```

Fig. 23.18 Creating and traversing a binary tree. (Part 2 of 2.)

Class **`Tree`** (lines 98–198 of Fig. 23.17) manipulates objects of class **`TreeNode`**. Class **`Tree`** has as **`private`** data **`root`** (line 100)—a reference to the root node of the tree. The class contains **`public`** method **`InsertNode`** (lines 111–121) to insert a new node in the tree and **`public`** methods **`PreorderTraversal`** (lines 124–130), **`InorderTraversal`** (lines 149–155) and **`PostorderTraversal`** (lines 174–180) to begin traversals of the tree. Each of these methods calls a separate recursive utility method to perform the traversal operations on the internal representation of the tree. The **`Tree`** constructor (lines 103–106) initializes **`root`** to **`null`** to indicate that the tree initially is empty.

The **`Tree`** class's method **`InsertNode`** (lines 67–74) first locks the **`Tree`** object for thread safety, then determines whether the tree is empty. If so, line 116 allocates a new **`TreeNode`**, initializes the node with the integer being inserted in the tree and assigns the new node to **`root`**. If the tree is not empty, **`InsertNode`** calls **`TreeNode`** method **`Insert`** (lines 67–93), which recursively determines the location for the new node in the tree and inserts the node at that location. *A node can be inserted only as a leaf node in a binary search tree.*

The **`TreeNode`** method **`Insert`** compares the value to insert with the **`data`** value in the root node. If the insert value is less than the root-node data, the program determines whether the left subtree is empty (line 73). If so, line 74 allocates a new **`TreeNode`**, initializes it with the integer being inserted and assigns the new node to reference **`leftNode`**. Otherwise, line 78 recursively calls **`Insert`** for the left subtree to insert the value into the left subtree. If the insert value is greater than the root-node data, the program determines whether the right subtree is empty (line 85). If so, line 86 allocates a new **`TreeNode`**, initializes it with the integer being inserted and assigns the new node to reference **`rightNode`**. Otherwise, line 90 recursively calls **`Insert`** for the right subtree to insert the value in the right subtree.

Methods **`InorderTraversal`**, **`PreorderTraversal`** and **`PostorderTraversal`** call helper methods **`InorderHelper`** (lines 158–171), **`PreorderHelper`** (lines 133–146) and **`PostorderHelper`** (lines 183–196), respectively, to traverse the tree and print the node values. The purpose of the helper methods in class **`Tree`** is to allow the programmer to start a traversal without the need to obtain a reference to the **`root`** node first, then call the recursive method with that reference. Methods **`InorderTraversal`**, **`PreorderTraversal`** and **`PostorderTraversal`** simply take the **`private`** reference **`root`** and pass it to the appropriate helper method to initiate a traversal of the tree. For the following discussion, we use the binary search tree shown in Fig. 23.19.

Method **`InorderHelper`** (lines 158–171) defines the steps for an inorder traversal. Those steps are as follows:

1. If the argument is **`null`**, return immediately.
2. Traverse the left subtree with a call to **`InorderHelper`** (line 164).
3. Process the value in the node (line 167).
4. Traverse the right subtree with a call to **`InorderHelper`** (line 170).

The inorder traversal does not process the value in a node until the values in that node's left subtree are processed. The inorder traversal of the tree in Fig. 23.19 is

```
6 13 17 27 33 42 48
```

Note that the inorder traversal of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually sorts the data—thus, this process is called the *binary tree sort*.

Method **`PreorderHelper`** (lines 133–146) defines the steps for a preorder traversal. Those steps are as follows:

1. If the argument is **`null`**, return immediately.
2. Process the value in the node (line 139).
3. Traverse the left subtree with a call to **`PreorderHelper`** (line 142).
4. Traverse the right subtree with a call to **`PreorderHelper`** (line 145).

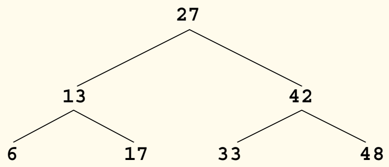


Fig. 23.19 A binary search tree.

The preorder traversal processes the value in each node as the node is visited. After processing the value in a given node, the preorder traversal processes the values in the left subtree, then the values in the right subtree. The preorder traversal of the tree in Fig. 23.19 is

```
27 13 6 17 42 33 48
```

Method **`PostorderHelper`** (lines 183–198) defines the steps for a postorder traversal. Those steps are as follows:

1. If the argument is **`null`**, return immediately.
2. Traverse the left subtree with a call to **`PostorderHelper`** (line 189).
3. Traverse the right subtree with a call to **`PostorderHelper`** (line 192).
4. Process the value in the node (line 195).

The postorder traversal processes the value in each node after the values of all that node's children are processed. The postorder traversal of the tree in Fig. 23.19 is

```
6 17 13 33 48 42 27
```

The binary search tree facilitates *duplicate elimination*. While building a tree, the insertion operation recognizes attempts to insert a duplicate value, because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the insertion operation eventually compares the duplicate with a node containing the same value. At this point, the insertion operation might simply discard the duplicate value.

Searching a binary tree for a value that matches a key value is fast, especially for *tightly packed* trees. In a tightly packed tree, each level contains about twice as many elements as the previous level. Figure 23.19 is a tightly packed binary tree. A binary search tree with n elements has a minimum of $\log_2 n$ levels. Thus, at most $\log_2 n$ comparisons are required either to find a match or to determine that no match exists. Searching a (tightly packed) 1000-element binary search tree requires at most 10 comparisons, because $2^{10} > 1000$. Searching a (tightly packed) 1,000,000-element binary search tree requires at most 20 comparisons, because $2^{20} > 1,000,000$.

The chapter exercises present algorithms for other binary tree operations, such as performing a *level-order traversal of a binary tree*. The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root-node level. On each level of the tree, a level-order traversal visits the nodes from left to right.

23.6.2 Binary Search Tree of `IComparable` Objects

The binary tree example in Section 23.6.1 works nicely when all the data is of type **`int`**. Suppose that you want to manipulate a binary tree of double values. You could rewrite the **`TreeNode`** and **`Tree`** classes with different names and customize the classes to manipulate double values. Similarly, for each data type you could create customized versions of classes **`TreeNode`** and **`Tree`**. This results in a proliferation of code, which can become difficult to manage and maintain. The C++ programming language provides a technology called templates that enables us to write a class definition once, then have the compiler generate new versions of the class for any data type we choose.

Ideally, we would like to define the functionality of a binary tree once and reuse that functionality for many data types. Languages like Java™ and C# provide polymorphic capabilities that enable all objects to be manipulated in a uniform manner. Using such capabilities enables us to design a more flexible data structure.

In our next example, we take advantage of C#'s polymorphic capabilities by implementing **`TreeNode`** and **`Tree`** classes that manipulate objects of any type that implements interface ***`IComparable`*** (namespace **`System`**). It is imperative that we be able to compare objects stored in a binary search, so we can determine the path to the insertion point of a new node. Classes that implement **`IComparable`** define method ***`CompareTo`***, which compares the object that invokes the method with the object that the method receives as an argument. The method returns an **`int`** value less than zero if the calling object is less than the argument object, zero if the objects are equal, a positive value if the calling object is greater than the argument object. Also, both the calling and argument objects must be of the same data type; otherwise, the method throws an **`ArgumentException`**.

The program of Fig. 23.20 and Fig. 23.21 enhances the program from Section 23.6.1 to manipulate **`IComparable`** objects. One restriction on the new versions of classes **`TreeNode`** and **`Tree`** in Fig. 23.20 is that each **`Tree`** object can contain objects of only one data type (e.g., all **`string`**s or all **`double`**s). If a program attempts to insert multiple data types in the same **`Tree`** object, **`ArgumentException`**s will occur. We modified only six lines of code in class **`TreeNode`** (lines 13, 17, 38, 67, 70 and 82) and one line of code in class **`Tree`** (line 111) to enable processing of **`IComparable`** objects. With the exception of lines 70 and 82, all other changes simply replaced the type **`int`** with the type **`IComparable`**. Lines 70 and 82 previously used the **`<`** and **`>`** operators to compare the value being inserted with the value in a given node. These lines now compare **`IComparable`** objects via the interface's method **`CompareTo`**, then test the method's return value to determine whether it is less than zero (the calling object is less than the argument object) or greater than zero (the calling object is greater than the argument object), respectively.

Class **`TreeTest`** (Fig. 23.21) creates three **`Tree`** objects to store **`int`**, **`double`** and **`string`** values, all of which the .NET Framework defines as **`IComparable`** types. The program populates the trees with the values in arrays **`intArray`** (line 15), **`doubleArray`** (lines 16–17) and **`stringArray`** (lines 18–19), respectively.

Method **`populateTree`** (lines 38–48) receives an **`Array`** containing the initializer values for the **`Tree`**, a **`Tree`** into which the array elements will be placed and a **`string`** representing the **`Tree`** name as arguments, then inserts each **`Array`** element in the **`Tree`**. Method **`traverseTree`** (lines 51–68) receives a **`Tree`** and a **`string`** representing the **`Tree`** name as arguments, then outputs the preorder, inorder and postorder traversals of the **`Tree`**. Note that the inorder traversal of each **`Tree`** outputs the data in sorted order regardless of the data type stored in the **`Tree`**. Our polymorphic implementation of class Tree invokes the appropriate data type's **`CompareTo`** method to determine the path to each value's insertion point by using the standard binary search tree insertion rules. Also, notice that the **`Tree`** of **`string`**s appears in alphabetical order.

```
1  // Fig. 23.20: BinaryTreeLibrary2.cs
2  // Definition of class TreeNode and class Tree for IComparable
3  // objects.
4
```

Fig. 23.20 Definitions of class **`TreeNode`** and **`Tree`** for manipulating **`IComparable`** objects. (Part 1 of 5.)

```
using System;

namespace BinaryTreeLibrary2
{
   // class TreeNode definition
   class TreeNode
   {
      private TreeNode leftNode;
      private IComparable data;
      private TreeNode rightNode;

      // initialize data and make this a leaf node
      public TreeNode( IComparable nodeData )
      {
         data = nodeData;
         leftNode = rightNode = null;  // node has no children
      }

      // LeftNode property
      public TreeNode LeftNode
      {
         get
         {
            return leftNode;
         }

         set
         {
            leftNode = value;
         }
      }

      // Data property
      public IComparable Data
      {
         get
         {
            return data;
         }

         set
         {
            data = value;
         }
      }

      // RightNode property
      public TreeNode RightNode
      {
         get
         {
```

Fig. 23.20 Definitions of class `TreeNode` and `Tree` for manipulating `IComparable` objects. (Part 2 of 5.)

```
               return rightNode;
            }

            set
            {
               rightNode = value;
            }
         }

         // insert TreeNode into Tree that contains nodes;
         // ignore duplicate values
         public void Insert( IComparable insertValue )
         {
            // insert in left subtree
            if ( insertValue.CompareTo( data ) < 0 )
            {
               // insert new TreeNode
               if ( leftNode == null )
                  leftNode = new TreeNode( insertValue );

               // continue traversing left subtree
               else
                  leftNode.Insert( insertValue );
            }

            // insert in right subtree
            else if ( insertValue.CompareTo( data ) > 0 )
            {
               // insert new TreeNode
               if ( rightNode == null )
                  rightNode = new TreeNode( insertValue );

               // continue traversing right subtree
               else
                  rightNode.Insert( insertValue );
            }

         }  // end method Insert

      }  // end class TreeNode

      // class Tree definition
      public class Tree
      {
         private TreeNode root;

         // construct an empty Tree of integers
         public Tree()
         {
            root = null;
         }

```

Fig. 23.20 Definitions of class **`TreeNode`** and **`Tree`** for manipulating **`IComparable`** objects. (Part 3 of 5.)

```
      // Insert a new node in the binary search tree.
      // If the root node is null, create the root node here.
      // Otherwise, call the insert method of class TreeNode.
      public void InsertNode( IComparable insertValue )
      {
         lock ( this )
         {
            if ( root == null )
               root = new TreeNode( insertValue );

            else
               root.Insert( insertValue );
         }
      }

      // begin preorder traversal
      public void PreorderTraversal()
      {
         lock ( this )
         {
            PreorderHelper( root );
         }
      }

      // recursive method to perform preorder traversal
      private void PreorderHelper( TreeNode node )
      {
         if ( node == null )
            return;

         // output node data
         Console.Write( node.Data + " " );

         // traverse left subtree
         PreorderHelper( node.LeftNode );

         // traverse right subtree
         PreorderHelper( node.RightNode );
      }

      // begin inorder traversal
      public void InorderTraversal()
      {
         lock ( this )
         {
            InorderHelper( root );
         }
      }

      // recursive method to perform inorder traversal
      private void InorderHelper( TreeNode node )
      {
```

Fig. 23.20 Definitions of class `TreeNode` and `Tree` for manipulating `IComparable` objects. (Part 4 of 5.)

```
         if ( node == null )
            return;

         // traverse left subtree
         InorderHelper( node.LeftNode );

         // output node data
         Console.Write( node.Data + " " );

         // traverse right subtree
         InorderHelper( node.RightNode );
      }

      // begin postorder traversal
      public void PostorderTraversal()
      {
         lock ( this )
         {
            PostorderHelper( root );
         }
      }

      // recursive method to perform postorder traversal
      private void PostorderHelper( TreeNode node )
      {
         if ( node == null )
            return;

         // traverse left subtree
         PostorderHelper( node.LeftNode );

         // traverse right subtree
         PostorderHelper( node.RightNode );

         // output node data
         Console.Write( node.Data + " " );
      }

   } // end class Tree
}
```

Fig. 23.20 Definitions of class `TreeNode` and `Tree` for manipulating `IComparable` objects. (Part 5 of 5.)

```
// Fig. 23.21: TreeTest.cs
// This program tests class Tree.

using System;
using BinaryTreeLibrary2;
```

Fig. 23.21 Demonstrating class Tree with `IComparable` objects. (Part 1 of 3.)

```
namespace TreeTest
{
   // class TreeTest definition
   public class TreeTest
   {
      // test class Tree
      static void Main( string[] args )
      {
         int[] intArray = { 8, 2, 4, 3, 1, 7, 5, 6 };
         double[] doubleArray =
            { 8.8, 2.2, 4.4, 3.3, 1.1, 7.7, 5.5, 6.6 };
         string[] stringArray = { "eight", "two", "four",
            "three", "one", "seven", "five", "six" };

         // create int Tree
         Tree intTree = new Tree();
         populateTree( intArray, intTree, "intTree" );
         traverseTree( intTree, "intTree" );

         // create double Tree
         Tree doubleTree = new Tree();
         populateTree( doubleArray, doubleTree, "doubleTree" );
         traverseTree( doubleTree, "doubleTree" );

         // create string Tree
         Tree stringTree = new Tree();
         populateTree( stringArray, stringTree, "stringTree" );
         traverseTree( stringTree, "stringTree" );
      }

      // populate Tree with array elements
      static void populateTree(
         Array array, Tree tree, string name )
      {
         Console.WriteLine( "\nInserting into " + name + ":" );

         foreach ( IComparable data in array )
         {
            Console.Write( data + " " );
            tree.InsertNode( data );
         }
      }

      // insert perform traversals
      static void traverseTree( Tree tree, string treeType )
      {
         // perform preorder traveral of tree
         Console.WriteLine(
            "\n\nPreorder traversal of " + treeType );
         tree.PreorderTraversal();
```

Fig. 23.21 Demonstrating class Tree with `IComparable` objects. (Part 2 of 3.)

```
         // perform inorder traveral of tree
         Console.WriteLine(
            "\n\nInorder traversal of " + treeType );
         tree.InorderTraversal();

         // perform postorder traveral of tree
         Console.WriteLine(
            "\n\nPostorder traversal of " + treeType );
         tree.PostorderTraversal();
         Console.WriteLine( "\n" );
      }

   } // end class TreeTest
}
```

```
Inserting into intTree:
8 2 4 3 1 7 5 6

Preorder traversal of intTree
8 2 1 4 3 7 5 6

Inorder traversal of intTree
1 2 3 4 5 6 7 8

Postorder traversal of intTree
1 3 6 5 7 4 2 8


Inserting into doubleTree:
8.8 2.2 4.4 3.3 1.1 7.7 5.5 6.6

Preorder traversal of doubleTree
8.8 2.2 1.1 4.4 3.3 7.7 5.5 6.6

Inorder traversal of doubleTree
1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8

Postorder traversal of doubleTree
1.1 3.3 6.6 5.5 7.7 4.4 2.2 8.8


Inserting into stringTree:
eight two four three one seven five six

Preorder traversal of stringTree
eight two four five three one seven six

Inorder traversal of stringTree
eight five four one seven six three two

Postorder traversal of stringTree
five six seven one three four two eight
```

Fig. 23.21 Demonstrating class Tree with `IComparable` objects. (Part 3 of 3.)

23.7 Collection Classes

The previous sections of this chapter discussed how to create and manipulate data structures. The discussion was “low level,” in the sense that we painstakingly created each element of each data structure dynamically with **`new`** and modified the data structures by directly manipulating their elements and references to their elements. In this section, we consider the prepackaged data-structure classes provided by the .NET Framework. These classes are known as *collection classes*—they store collections of data. Each instance of one of these classes is known as a *collection*, which is a set of items.

With collection classes, instead of creating data structures, the programmer simply uses existing data structures, without concern for how the data structures are implemented. This methodology is a marvelous example of code reuse. Programmers can code faster and can expect excellent performance, maximizing execution speed and minimizing memory consumption.

Some examples of collections are the cards you hold in a card game, your favorite songs stored in your computer and the real-estate records in your local registry of deeds (which map book numbers and page numbers to property owners). The .NET Framework provides several collections. We demonstrate four collection classes—***`Array`***, ***`ArrayList`***, ***`Stack`*** and ***`Hashtable`***—all from namespace **`System.Collections`**, plus built-in array capabilities. In addition, namespace **`System.Collections`** provides several other data structures, including ***`BitArray`*** (a collection of true/false values), ***`Queue`*** and ***`SortedList`*** (a collection of key/value pairs that are sorted by key and can be accessed either by key or by index).

The .NET Framework provides ready-to-go, reusable components; you do not need to write your own collection classes. The collections are standardized so applications can share them easily, without having to be concerned with the details of their implementation. These collections are written for broad reuse. They are tuned for rapid execution and for efficient use of memory. As new data structures and algorithms are developed that fit this framework, a large base of programmers already will be familiar with the interfaces and algorithms implemented by those data structures.

23.7.1 Class `Array`

Chapter 7 presented basic array-processing capabilities, and many subsequent chapters used the techniques shown there. We discussed briefly that all arrays inherit from class **`Array`** (namespace **`System`**) which defines a **`Length`** property that specifies the number of elements in an array. In addition, class **`Array`** provides **`static`** methods that provide algorithms for processing arrays. Typically, class **`Array`** overloads these methods to provide multiple options for performing algorithms. For example, **`Array`** method **`Reverse`** can reverse the order of the elements in an entire array or can reverse the elements in a specified range of elements in an array. For a complete list of class **`Array`**’s **`static`** methods and their overloaded versions, see the online documentation for the class. Figure 23.22 demonstrates several **`static`** methods of class **`Array`**.

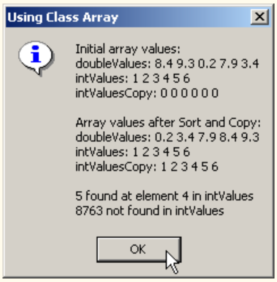
Line 28 uses **`static Array`** method ***`Sort`*** to sort an array of **`double`** values. When this method returns, the array contains its original elements sorted in ascending order.

Lines 31–32 uses **`static Array`** method ***`Copy`*** to copy elements from array **`intArray`** into array **`intArrayCopy`**. The first argument is the array to copy (**`intValues`**), the second argument is the destination array (**`intValuesCopy`**) and the third argument is an integer representing the number of elements to copy (in this case, **`intValues.Length`** specifies all elements).

```
// Fig. 23.22: UsingArray.cs
// Using Array class to perform common array manipulations.

using System;
using System.Windows.Forms;
using System.Collections;

namespace UsingArray
{
   // demonstrate algorithms of class Array
   class UsingArray
   {
      private int[] intValues = { 1, 2, 3, 4, 5, 6 };
      private double[] doubleValues =
         { 8.4, 9.3, 0.2, 7.9, 3.4 };
      private int[] intValuesCopy;
      private string output;

      // method to build and display program output
      public void Start()
      {
         intValuesCopy = new int[ intValues.Length ];

         output = "Initial array values:\n";
         PrintArray();  // output initial array contents

         // sort doubleValues
         Array.Sort( doubleValues );

         // copy intValues into intValuesCopy
         Array.Copy( intValues, intValuesCopy,
            intValues.Length );

         output += "\nArray values after Sort and Copy:\n";
         PrintArray();  // output array contents
         output += "\n";

         // search for 5 in intValues
         int result = Array.BinarySearch( intValues, 5 );
         output +=
            ( result >= 0 ? "5 found at element " + result :
                 "5 not found" ) + " in intValues\n";

         // search for 8763 in intValues
         result = Array.BinarySearch( intValues, 8763 );
         output +=
            ( result >= 0 ? "8763 found at element " + result :
                 "8763 not found" ) + " in intValues";
```

Fig. 23.22 Program that demonstrates class **`Array`**. (Part 1 of 2.)

```
49
50          MessageBox.Show( output, "Using Class Array",
51             MessageBoxButtons.OK, MessageBoxIcon.Information );
52       }
53
54       // append array content to output string
55       private void PrintArray()
56       {
57          output += "doubleValues: ";
58
59          foreach ( double element in doubleValues )
60             output += element + " ";
61
62          output += "\nintValues: ";
63
64          foreach ( int element in intValues )
65             output += element + " ";
66
67          output += "\nintValuesCopy: ";
68
69          foreach ( int element in intValuesCopy )
70             output += element + " ";
71
72          output += "\n";
73       }
74
75       // main entry point for application
76       static void Main( string[] args )
77       {
78          UsingArray application = new UsingArray();
79
80          application.Start();
81       }
82
83    } // end class UsingArray
84 }
```



```
Initial array values:
doubleValues: 8.4 9.3 0.2 7.9 3.4
intValues: 1 2 3 4 5 6
intValuesCopy: 0 0 0 0 0 0

Array values after Sort and Copy:
doubleValues: 0.2 3.4 7.9 8.4 9.3
intValues: 1 2 3 4 5 6
intValuesCopy: 1 2 3 4 5 6

5 found at element 4 in intValues
8763 not found in intValues
```

Fig. 23.22 Program that demonstrates class `Array`. (Part 2 of 2.)

Lines 39 and 45 invoke `static` `Array` method ***`BinarySearch`*** to perform binary searches on array `intValues`. Method `BinarySearch` receives the *sorted* array in which to search and the key for which to search. The method returns the index in the array at which it finds the key (but a negative number if the key was not found).

Other `static` `Array` methods include *`Clear`* (to set a range of elements to 0 or null), *`CreateInstance`* (to create a new array of a specified data type), *`IndexOf`* (to locate the first occurrence of an object in an array or portion of an array), *`LastIndexOf`* (to locate the last occurrence of an object in an array or portion of an array) and *`Reverse`* (to reverse the contents of an array or portion of an array).

23.7.2 Class `ArrayList`

In most programming languages, conventional arrays have a fixed size—they cannot be changed dynamically to conform to an application's execution-time memory requirements. In some applications, this fixed-size limitation presents a problem for programmers. They must choose between using fixed-size arrays that are large enough to store the maximum number of elements the program may require and using dynamic data structures that can grow and shrink the amount of memory required to store data in response to the changing requirements of a program at execution time.

The .NET Framework's class *`ArrayList`* collection mimics the functionality of conventional arrays and provides dynamic resizing of the collection through the class's methods. At any time an `ArrayList` contains a certain number of elements less than or equal to its *capacity*—the number of elements currently reserved for an `ArrayList`. A program can manipulate the capacity with `ArrayList` property `Capacity`. If an `ArrayList` needs to grow, it by default doubles its current `Capacity`.

Performance Tip 23.6

As with linked lists, inserting additional elements into an `ArrayList` whose current size is less than its capacity is a fast operation.

Performance Tip 23.7

It is a slow operation to insert an element into an `ArrayList` that needs to grow larger to accommodate a new element.

Performance Tip 23.8

If storage is at a premium, use method `TrimToSize` of class `ArrayList` to trim an `ArrayList` to its exact size. This will optimize an `ArrayList`'s memory use. Be careful—if the program needs to insert additional elements, the process will be slower because the `ArrayList` must grow dynamically (trimming leaves no room for growth).

Performance Tip 23.9

The default capacity increment, doubling the size of the `ArrayList`, may seem to waste storage, but doubling is an efficient way for an `ArrayList` to grow quickly to "about the right size." This is a much more efficient use of time than growing the `ArrayList` by one element at a time in response to insert operations.

`ArrayList`s store references to `object`s. All classes derive from class `Object`, so an `ArrayList` can contain objects of any type. Figure 23.23 lists some useful methods of class `ArrayList`.

Figure 23.24 demonstrates class `ArrayList` and several of its methods. Users can type a `string` into the user interface's `TextBox`, then press a button representing an `ArrayList` method to see that method's functionality. A `TextBox` displays messages indicating each operation's results.

Method	Description
`Add`	Adds an `object` to the `ArrayList`. Returns an `int` specifying the index at which the `object` was added.
`Clear`	Removes all the elements from the `ArrayList`.
`Contains`	Returns `true` if the specified `object` is in the `ArrayList`; otherwise, returns `false`.
`IndexOf`	Returns the index of the first occurrence of the specified `object` in the `ArrayList`.
`Insert`	Inserts an `object` at the specified index.
`Remove`	Removes the first occurrence of the specified `object`.
`RemoveAt`	Removes an object at the specified index.
`RemoveRange`	Removes a specified number of elements starting at a specified index in the `ArrayList`.
`Sort`	Sorts the `ArrayList`.
`TrimToSize`	Sets the `Capacity` of the `ArrayList` to be the number of elements the `ArrayList` currently contains.

Fig. 23.23 Some methods of class `ArrayList`.

```
// Fig. 23.24: ArrayListTest.cs
// Using class ArrayList.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Text;

namespace ArrayListTest
{
   // demonstrating ArrayList functionality
   public class ArrayListTest : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Button addButton;
      private System.Windows.Forms.TextBox inputTextBox;
      private System.Windows.Forms.Label inputLabel;
      private System.Windows.Forms.Button removeButton;
      private System.Windows.Forms.Button firstButton;
      private System.Windows.Forms.Button lastButton;
      private System.Windows.Forms.Button isEmptyButton;
      private System.Windows.Forms.Button containsButton;
      private System.Windows.Forms.Button locationButton;
```

Fig. 23.24 Demomstrating the `ArrayList` class. (Part 1 of 5.)

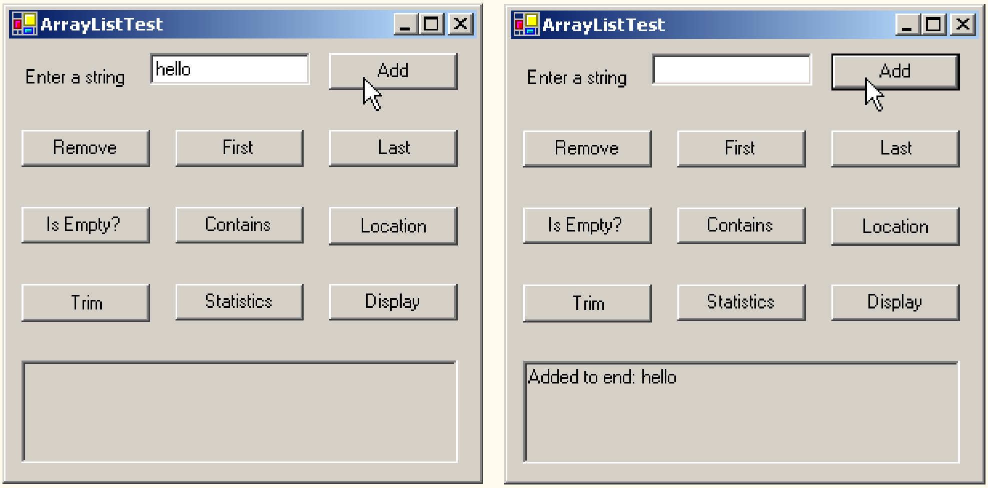
```
         private System.Windows.Forms.Button trimButton;
         private System.Windows.Forms.Button statisticsButton;
         private System.Windows.Forms.Button displayButton;

         // Required designer variable.
         private System.ComponentModel.Container components = null;
         private System.Windows.Forms.TextBox consoleTextBox;

         // ArrayList for manipulating strings
         private ArrayList arrayList = new ArrayList( 1 );

         public ArrayListTest()
         {
            // Required for Windows Form Designer support
            InitializeComponent();
         }

         // Visual Studio.NET generated code

         // main entry point for the application
         [STAThread]
         static void Main()
         {
            Application.Run( new ArrayListTest() );
         }

         // add item to end of arrayList
         private void addButton_Click(
            object sender, System.EventArgs e )
         {
            arrayList.Add( inputTextBox.Text );
            consoleTextBox.Text =
               "Added to end: " + inputTextBox.Text;
            inputTextBox.Clear();
         }

         // remove specified item from arrayList
         private void removeButton_Click(
            object sender, System.EventArgs e )
         {
            arrayList.Remove( inputTextBox.Text );
            consoleTextBox.Text = "Removed: " + inputTextBox.Text;
            inputTextBox.Clear();
         }

         // display first element
         private void firstButton_Click(
            object sender, System.EventArgs e )
         {
            // get first element
            try
            {
```

Fig. 23.24 Demomstrating the `ArrayList` class. (Part 2 of 5.)

```
            consoleTextBox.Text =
               "First element: " + arrayList[ 0 ];
         }

         // show exception if no elements in arrrayList
         catch ( ArgumentOutOfRangeException outOfRange )
         {
            consoleTextBox.Text = outOfRange.ToString();
         }
      }

      // display last element
      private void lastButton_Click(
         object sender, System.EventArgs e )
      {
         // get last element
         try
         {
            consoleTextBox.Text = "Last element: " +
               arrayList[ arrayList.Count - 1 ];
         }

         // show exception if no elements in arrrayList
         catch ( ArgumentOutOfRangeException outOfRange )
         {
            consoleTextBox.Text = outOfRange.ToString();
         }
      }

      // determine whether arrayList is empty
      private void isEmptyButton_Click(
         object sender, System.EventArgs e )
      {
         consoleTextBox.Text = ( arrayList.Count == 0 ?
            "arrayList is empty" : "arrayList is not empty" );
      }

      // determine whether arrayList contains specified object
      private void containsButton_Click(
         object sender, System.EventArgs e )
      {
         if ( arrayList.Contains( inputTextBox.Text ) )
            consoleTextBox.Text = "arrayList contains " +
               inputTextBox.Text;
         else
            consoleTextBox.Text = inputTextBox.Text +
               " not found";
      }

      // determine location of specified object
      private void locationButton_Click(
         object sender, System.EventArgs e )
      {
```

Fig. 23.24 Demomstrating the `ArrayList` class. (Part 3 of 5.)

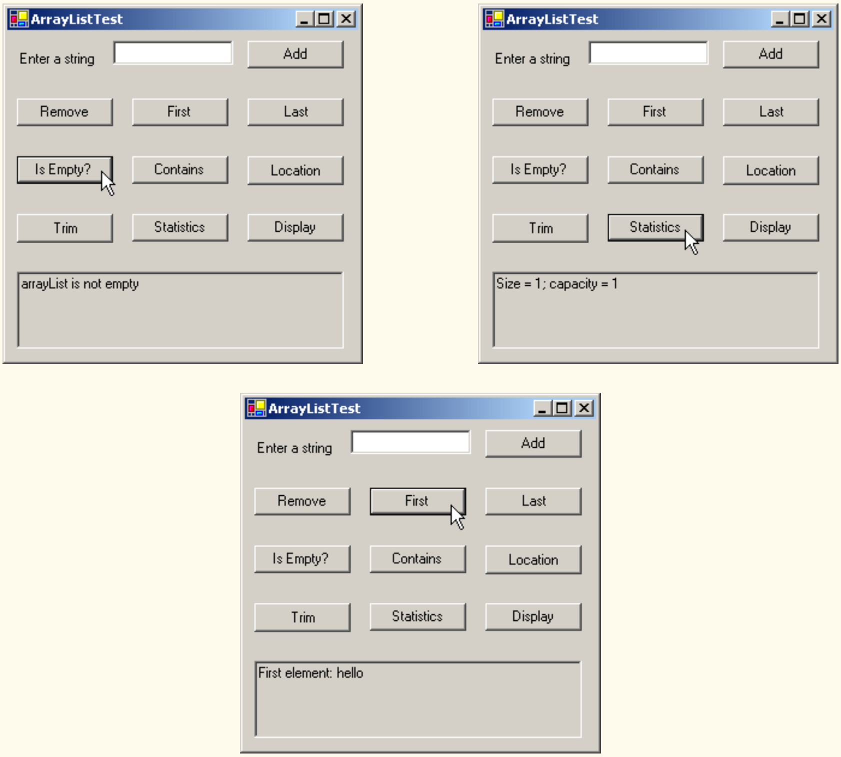
```
         consoleTextBox.Text = "Element is at location " +
            arrayList.IndexOf( inputTextBox.Text );
      }

      // trim arrayList to current size
      private void trimButton_Click(
         object sender, System.EventArgs e )
      {
         arrayList.TrimToSize();
         consoleTextBox.Text = "Vector trimmed to size";
      }

      // show arrayList current size and capacity
      private void statisticsButton_Click(
         object sender, System.EventArgs e )
      {
         consoleTextBox.Text = "Size = " + arrayList.Count +
            "; capacity = " + arrayList.Capacity;
      }

      // display contents of arrayList
      private void displayButton_Click(
         object sender, System.EventArgs e )
      {
         IEnumerator enumerator = arrayList.GetEnumerator();
         StringBuilder buffer = new StringBuilder();

         while ( enumerator.MoveNext() )
            buffer.Append( enumerator.Current + "  " );

         consoleTextBox.Text = buffer.ToString();
      }
   }
}
```


ArrayListTest
Enter a string
Add
Remove
First
Last
Is Empty?
Contains
Location
Trim
Statistics
Display
ArrayListTest
Enter a string
Add
Remove
First
Last
Is Empty?
Contains
Location
Trim
Statistics
Display

hello

Added to end: hello

Fig. 23.24 Demomstrating the `ArrayList` class. (Part 4 of 5.)



```
arrayList is not empty
```

```
Size = 1; capacity = 1
```

```
First element: hello
```

Fig. 23.24 Demomstrating the `ArrayList` class. (Part 5 of 5.)

The `ArrayList` in this example stores `string`s that users input in the `TextBox`. Line 35 creates an `ArrayList` with an initial capacity of one element. This `ArrayList` will double in size each time the user fills the array and attempts to add another element.

`ArrayList` method *`Add`* appends a new element at the end of an `ArrayList`. When the user clicks **Add**, event handler `addButton_Click` (lines 53–60) invokes method `Add` (line 56) to append the `string` in the `inputTextBox` to the `ArrayList`.

`ArrayList` method *`Remove`* deletes a specified item from an `ArrayList`. When the user clicks **Remove**, event handler `removeButton_Click` (line 63–69) invokes `Remove` (line 66) to remove the `string` specified in the `inputTextBox` from the `ArrayList`. If the object passed to `Remove` is in the `ArrayList`, the first occurrence of that object is removed, and all subsequent elements shift toward the beginning of the `ArrayList` to fill the empty position.

A program can access `ArrayList` elements like conventional array elements by following the `ArrayList` reference name with the array subscript operator (`[]`) and the desired index of the element. Event handlers `firstButton_Click` (lines 72–87) and `lastButton_Click` (lines 90–105) use the `ArrayList` subscript operator to retrieve the first element (line 79) and last element (line 97), respectively. An `ArgumentOutOfRangeException` occurs if the specified index is not both greater than 0 and less than the number of elements currently stored in the `ArrayList`.

Event handler `isEmptyButton_Click` (lines 108–113) uses `ArrayList` property *`Count`* (line 111) to determine whether the `ArrayList` is empty. Event handler `containsButton_Click` (lines 116–125) uses `ArrayList` method `Contains` (line 119) to determine whether the the given object is currently in the `ArrayList`. If so, the method returns `true`; otherwise, it returns `false`.

Performance Tip 23.10

`ArrayList` method `Contains` performs a linear search, which is a costly operation for large `ArrayList`s. If the `ArrayList` is sorted, use `ArrayList` method `BinarySearch` to perform a more efficient search.

When the user clicks **Location**, event handler `locationButton_Click` (lines 128–133) invokes `ArrayList` method *`IndexOf`* (line 132) to determine the index of a particular object in the `ArrayList`. `IndexOf` returns `-1` if the element is not found.

When the user clicks **Trim**, event handler `trimButton_Click` (lines 136–141) invokes method *`TrimToSize`* (line 139) to set the *`Capacity`* property to equal the `Count` property. This reduces the storage capacity of the `ArrayList` to the exact number of elements currently in the `ArrayList`.

When the user clicks **Statistics**, `statisticsButton_Click` (lines 144–149) uses the `Count` and `Capacity` properties to display the current number of elements in the `ArrayList` and the maximum number of elements that can be stored without allocating more memory to the `ArrayList`.

When users click **Display**, `displayButton_Click` (lines 152–162) outputs the contents of the `ArrayList`. This event handler uses an *`IEnumerator`* (sometimes called an *enumerator* or an *iterator*) to traverse the elements of an `ArrayList` one element at a time. Interface `IEnumerator` defines methods *`MoveNext`* and *`Reset`* and property *`Current`*. `MoveNext` moves the enumerator to the next element in the `ArrayList`. The first call to `MoveNext` positions the enumerator at the first element of the `ArrayList`. `MoveNext` returns `true` if there is at least one more element in the `ArrayList`; otherwise, the method returns `false`. Method `Reset` positions the enumerator before the first element of the `ArrayList`. Methods `MoveNext` and `Reset` throw an `InvalidOperationException` if the contents of the collection are modified in any way after the enumerator's creation. Property `Current` returns the object at the current location in the `ArrayList`.

Line 155 creates an `IEnumerator` called `enumerator` and assigns it the result of calling `ArrayList` method *`GetEnumerator`*. Lines 158–159 iterate while `MoveNext` returns `true`, retrieve the current item via property `Count` and append it to `buffer`. When the loop terminates, line 161 displays the contents of `buffer`.

23.7.3 Class `Stack`

The `Stack` class, as its name implies, implements a stack data structure. This class provides much of the functionality that we defined in our implementation in Section 23.4. Refer back to that section for a discussion of stack data structure concepts. The application in Fig. 23.25 provides a GUI that enables the user to test many `Stack` methods. Line 38 of the `StackTest` constructor creates a `Stack` with the default initial capacity (10 elements).

As one might expect, class `Stack` has methods `Push` and `Pop` to perform the basic stack operations. Method `Push` takes an `object` as an argument and adds it to the top of the **`Stack`**. If the number of items on the **`Stack`** (the **`Count`** property) is equal to the capacity at the time of the **`Push`** operation, the **`Stack`** grows to accommodate more **`object`**s. Event handler **`pushButton_Click`** (lines 51–56) uses method **`Push`** to add a user-specified string to the stack (line 54).

Method **`Pop`** takes no arguments. This method removes and returns the object currently on top of the **`Stack`**. Event handler **`popButton_Click`** (lines 59–73) calls method **`Pop`** (line 57) to remove an object from the **`Stack`**. An **`InvalidOperationException`** occurs if the **`Stack`** is empty when the program calls **`Pop`**.

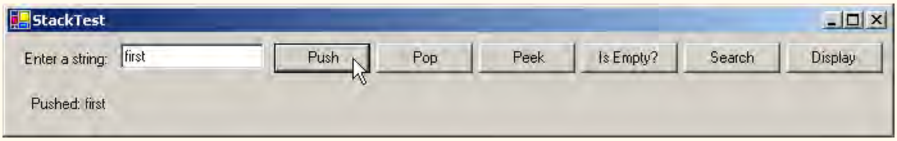
```
 1  // Fig. 23.25: StackTest.cs
 2  // Demonstrates class Stack of namespace System.Collections.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10  using System.Text;
11
12  namespace StackTest
13  {
14     // demonstrate Stack collection
15     public class StackTest : System.Windows.Forms.Form
16     {
17        private System.Windows.Forms.Label inputLabel;
18        private System.Windows.Forms.TextBox inputTextBox;
19        private System.Windows.Forms.Button pushButton;
20        private System.Windows.Forms.Button popButton;
21        private System.Windows.Forms.Button peekButton;
22        private System.Windows.Forms.Button isEmptyButton;
23        private System.Windows.Forms.Button searchButton;
24        private System.Windows.Forms.Button displayButton;
25        private System.Windows.Forms.Label statusLabel;
26
27        // Required designer variable.
28        private System.ComponentModel.Container components = null;
29
30        private Stack stack;
31
32        public StackTest()
33        {
34           // Required for Windows Form Designer support
35           InitializeComponent();
36
37           // create Stack
38           stack = new Stack();
39        }
40
41        // Visual Studio.NET generated code
42
```

Fig. 23.25 Using the **`Stack`** class. (Part 1 of 4.)

```
        // main entry point for the application
        [STAThread]
        static void Main()
        {
           Application.Run( new StackTest() );
        }

        // push element onto stack
        private void pushButton_Click(
           object sender, System.EventArgs e )
        {
           stack.Push( inputTextBox.Text );
           statusLabel.Text = "Pushed: " + inputTextBox.Text;
        }

        // pop element from stack
        private void popButton_Click(
           object sender, System.EventArgs e )
        {
           // pop element
           try
           {
              statusLabel.Text = "Popped: " + stack.Pop();
           }

           // print message if stack is empty
           catch ( InvalidOperationException invalidOperation )
           {
              statusLabel.Text = invalidOperation.ToString();
           }
        }

        // peek at top element of stack
        private void peekButton_Click(
           object sender, System.EventArgs e )
        {
           // view top element
           try
           {
              statusLabel.Text = "Top: " + stack.Peek();
           }

           // print message if stack is empty
           catch ( InvalidOperationException invalidOperation )
           {
              statusLabel.Text = invalidOperation.ToString();
           }
        }

        // determine whether stack is empty
        private void isEmptyButton_Click(
           object sender, System.EventArgs e )
        {
```

Fig. 23.25 Using the `Stack` class. (Part 2 of 4.)

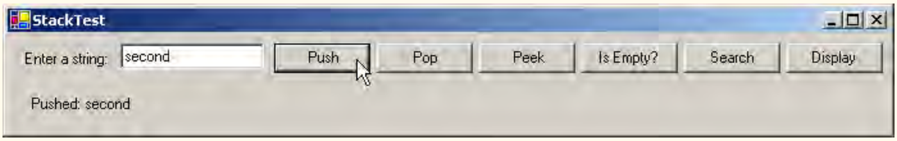
```
         statusLabel.Text = ( stack.Count == 0 ?
            "Stack is empty" : "Stack is not empty" );
      }

      // determine whether specified element is on stack
      private void searchButton_Click(
         object sender, System.EventArgs e )
      {
         string result = stack.Contains( inputTextBox.Text ) ?
            " found" : " not found";

         statusLabel.Text = inputTextBox.Text + result;
      }

      // display stack contents
      private void displayButton_Click(
         object sender, System.EventArgs e )
      {
         IEnumerator enumerator = stack.GetEnumerator();
         StringBuilder buffer = new StringBuilder();

         // while the enumerator can move on to the next element
         // print that element out.
         while ( enumerator.MoveNext() )
            buffer.Append( enumerator.Current + " " );

         statusLabel.Text = buffer.ToString();
      }
   }
}
```



Enter a string: first

Pushed: first

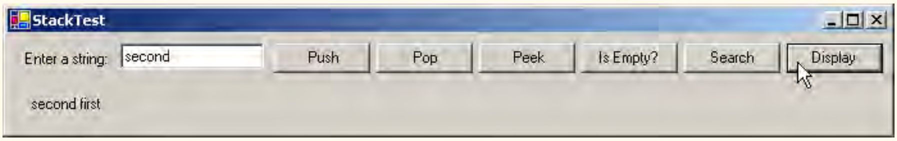
StackTest
Push
Pop
Peek
Is Empty?
Search
Display



Enter a string: second

Pushed: second

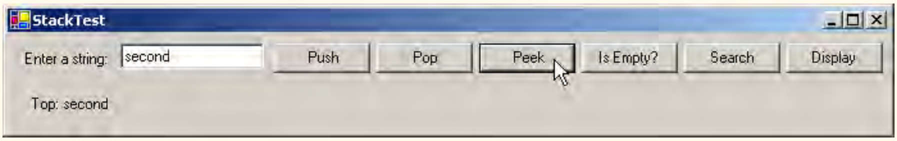
StackTest
Push
Pop
Peek
Is Empty?
Search
Display



Enter a string: second

second first

StackTest
Push
Pop
Peek
Is Empty?
Search
Display

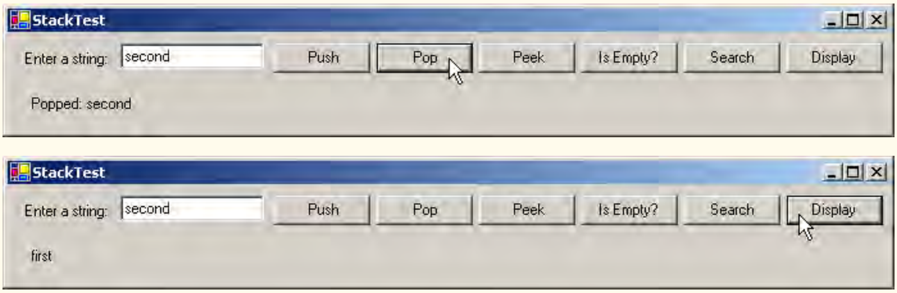


Enter a string: second

Top: second

StackTest
Push
Pop
Peek
Is Empty?
Search
Display

Fig. 23.25 Using the `Stack` class. (Part 3 of 4.)



Enter a string: second

Popped: second

Enter a string: second

first

Fig. 23.25 Using the **`Stack`** class. (Part 4 of 4.)

Method **`Peek`** returns the value of the top stack element, but does not remove the element from the **`Stack`**. We demonstrate **`Peek`** at line 82 in event handler **`peekButton_Click`** (lines 76–90) to view the object on top of the **`Stack`**. As with **`Pop`**, an **`InvalidOperationException`** occurs if the **`Stack`** is empty when the program calls **`Peek`**.

Common Programming Error 23.3

*Attempting to **`Peek`** or **`Pop`** an empty **`Stack`** (a **`Stack`** whose **`Count`** property equals 0) causes an **`InvalidOperationException`**.*

Event handler **`isEmptyButton_Click`** (lines 93–98) determines whether the **`Stack`** is empty by comparing the **`Stack`**'s **`Count`** property to 0. If it is 0, the **`Stack`** is empty; otherwise, it is not. Event handler **`searchButton_Click`** (lines 101–108) uses **`Stack`** method **`Contains`** (lines 104–105) to determine whether the **`Stack`** contains the object specified as its argument. **`Contains`** returns **`true`** if the **`Stack`** contains the specified object, **`false`** otherwise.

Event handler **`isEmptyButton_Click`** (lines 111–123) uses an IEnumerator to traverse the **`Stack`** and display its contents.

23.7.4 Class `Hashtable`

Object-oriented programming languages facilitate creating new types. When a program creates objects of new or existing types, it then needs to manage those objects efficiently. This includes sorting and retrieving objects. Sorting and retrieving information with arrays is efficient if some aspect of your data directly matches the key value and if those keys are unique and tightly packed. If you have 100 employees with nine-digit Social Security numbers and you want to store and retrieve employee data by using the Social Security number as a key, it would nominally require an array with 999,999,999 elements, because there are 999,999,999 unique nine-digit numbers. This is impractical for virtually all applications that key on Social Security numbers. If you could have an array that large, you could get very high performance storing and retrieving employee records by simply using the Social Security number as the array index.

A large variety of applications have this problem—namely, that either the keys are of the wrong type (i.e., not nonnegative integers), or they are of the right type, but they are sparsely spread over a large range.

What is needed is a high-speed scheme for converting keys such as Social Security numbers and inventory part numbers into unique array subscripts. Then, when an application needs to store something, the scheme could convert the application key rapidly into a subscript and the record of information could be stored at that location in the array. Retrieval occurs the same way—once the application has a key for which it wants to retrieve the data record, the application simply applies the conversion to the key, which produces the array subscript where the data resides in the array and retrieves the data.

The scheme we describe here is the basis of a technique called *hashing*. Why the name? Because, when we convert a key into an array subscript, we literally scramble the bits, forming a kind of “mishmash” number. The number actually has no real significance beyond its usefulness in storing and retrieving this particular data record.

A glitch in the scheme occurs when *collisions* occur [i.e., two different keys “hash into” the same cell (or element) in the array]. Since we cannot sort two different data records into the same space, we need to find an alternative home for all records beyond the first that hash to a particular array subscript. Many schemes exist for doing this. One is to “hash again” (i.e., to reapply the hashing transformation to the key to provide a next candidate cell in the array). The hashing process is designed to be quite random, so the assumption is that with just a few hashes, an available cell will be found.

Another scheme uses one hash to locate the first candidate cell. If the cell is occupied, successive cells are searched linearly until an available cell is found. Retrieval works the same way—the key is hashed once, the resulting cell is checked to determine whether it contains the desired data. If it does, the search is complete. If it does not, successive cells are searched linearly until the desired data is found.

The most popular solution to hash-table collisions is to have each cell of the table be a hash “bucket,” typically a linked list of all the key/value pairs that hash to that cell. This is the solution that the .NET Framework’s ***Hashtable*** class implements.

The *load factor* is one factor that affects the performance of hashing schemes. The load factor is the ratio of the number of occupied cells in the hash table to the size of the hash table. The closer the ratio gets to 1.0, the greater the chance of collisions.

Performance Tip 23.11

The load factor in a hash table is a classic example of a space/time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower due to increased hashing collisions. By decreasing the load factor, we get better program speed because of reduced hashing collisions, but we get poorer memory utilization because a larger portion of the hash table remains empty.

Programming hash tables properly is too complex for most casual programmers. Computer science students study hashing schemes thoroughly in courses called “Data Structures” and “Algorithms.” Recognizing the value of hashing, C# provides class **Hashtable** and some related features to enable programmers to take advantage of hashing without the complex details.

The preceding sentence is profoundly important in our study of object-oriented programming. Classes encapsulate and hide complexity (i.e., implementation details) and offer user-friendly interfaces. Crafting classes to do this properly is one of the most valued skills in the field of object-oriented programming.

A *hash function* performs a calculation that determines where to place data in the hashtable. The hash function is applied to the key in a key/value pair of objects. Class **Hash-**

`table` can accept any object as a key. For this reason, class **`Object`** defines method **`GetHashCode`**, which all objects in C# inherit. Most classes that are candidates to be used as keys in a hash table override this method to provide one that performs efficient hashcode calculations for a specific data type. For example, a **`string`** has a hashcode calculation that is based on the contents of the **`string`**. Figure 23.26 demonstrates several methods of class **`Hashtable`**.

```
// Fig. 23.26: HashtableTest.cs
// Demonstrate class Hashtable of namespace System.Collections.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Text;

namespace HashTableTest
{
   // demonstrate Hashtable functionality
   public class HashTableTest : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Label firstNameLabel;
      private System.Windows.Forms.Label lastNameLabel;
      private System.Windows.Forms.Button addButton;
      private System.Windows.Forms.TextBox lastNameTextBox;
      private System.Windows.Forms.TextBox consoleTextBox;
      private System.Windows.Forms.TextBox firstNameTextBox;
      private System.Windows.Forms.Button getButton;
      private System.Windows.Forms.Button removeButton;
      private System.Windows.Forms.Button emptyButton;
      private System.Windows.Forms.Button containsKeyButton;
      private System.Windows.Forms.Button clearTableButton;
      private System.Windows.Forms.Button listObjectsButton;
      private System.Windows.Forms.Button listKeysButton;
      private System.Windows.Forms.Label statusLabel;

      // Required designer variable.
      private System.ComponentModel.Container components = null;

      // Hashtable to demonstrate functionality
      private Hashtable table;

      public HashTableTest()
      {
         // Required for Windows Form Designer support
         InitializeComponent();

         // create Hashtable object
         table = new Hashtable();
      }
```

Fig. 23.26 Using the **`Hashtable`** class. (Part 1 of 5.)

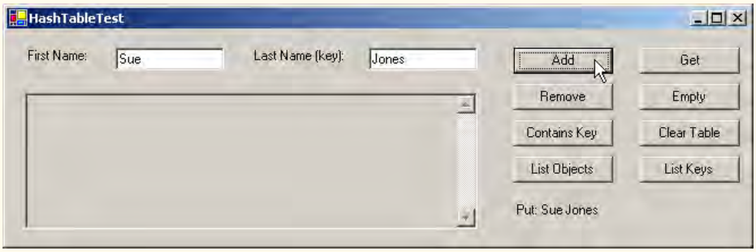
```

      // Visual Studio.NET generated code

      // main entry point for the application
      [STAThread]
      static void Main()
      {
         Application.Run( new HashTableTest() );
      }

      // add last name and Employee object to table
      private void addButton_Click(
         object sender, System.EventArgs e )
      {
         Employee employee = new Employee( firstNameTextBox.Text,
            lastNameTextBox.Text );

         // add new key/value pair
         try
         {
            table.Add( lastNameTextBox.Text, employee );
            statusLabel.Text = "Put: " + employee.ToString();
         }

         // if key is null or already in table, output message
         catch ( ArgumentException argumentException )
         {
            statusLabel.Text = argumentException.ToString();
         }
      }

      // get object for given key
      private void getButton_Click(
         object sender, System.EventArgs e )
      {
         object result = table[ lastNameTextBox.Text ];

         if ( result != null )
            statusLabel.Text = "Get: " + result.ToString();
         else
            statusLabel.Text = "Get: " + lastNameTextBox.Text +
               " not in table";
      }

      // remove key/value pair from table
      private void removeButton_Click(
         object sender, System.EventArgs e )
      {
         table.Remove( lastNameTextBox.Text );
         statusLabel.Text = "Object Removed";
      }

```

Fig. 23.26 Using the `Hashtable` class. (Part 2 of 5.)

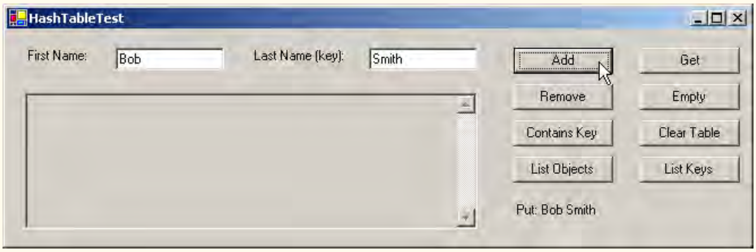
```
      // determine whether table is empty
      private void emptyButton_Click(
         object sender, System.EventArgs e )
      {
         statusLabel.Text = "Table is " + (
            table.Count == 0 ? "empty" : "not empty" );
      }

      // determine whether table contains specified key
      private void containsKeyButton_Click(
         object sender, System.EventArgs e )
      {
         statusLabel.Text = "Contains key: " +
            table.ContainsKey( lastNameTextBox.Text );
      }

      // discard all table contents
      private void clearTableButton_Click(
         object sender, System.EventArgs e )
      {
         table.Clear();
         statusLabel.Text = "Clear: Table is now empty";
      }

      // display list of objects in table
      private void listObjectsButton_Click(
         object sender, System.EventArgs e )
      {
         IDictionaryEnumerator enumerator =
            table.GetEnumerator();
         StringBuilder buffer = new StringBuilder();

         while ( enumerator.MoveNext() )
            buffer.Append( enumerator.Value + "\r\n" );

         consoleTextBox.Text = buffer.ToString();
      }

      // display list of keys in table
      private void listKeysButton_Click(
         object sender, System.EventArgs e )
      {
         IDictionaryEnumerator enumerator =
            table.GetEnumerator();
         StringBuilder buffer = new StringBuilder();

         while ( enumerator.MoveNext() )
            buffer.Append( enumerator.Key + "\r\n" );

         consoleTextBox.Text = buffer.ToString();
      }

   } // end class HashtableTest
```

Fig. 23.26 Using the **`Hashtable`** class. (Part 3 of 5.)

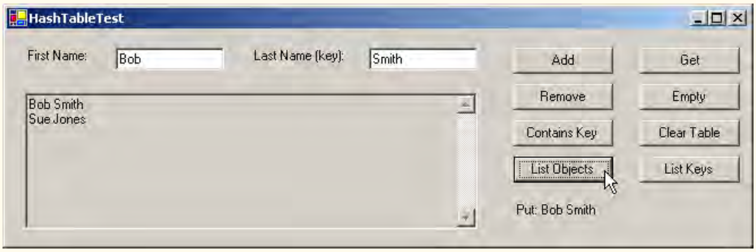
```

   // class Employee for use with HashtableTest
   class Employee
   {
      private string first, last;

      // constructor
      public Employee( string fName, string lName )
      {
         first = fName;
         last = lName;
      }

      // return Employee first and last names as string
      public override string ToString()
      {
         return first + " " + last;
      }

   } // end class Employee
}
```


HashTableTest
Add
Get
Remove
Empty
Contains Key
Clear Table
List Objects
List Keys

First Name: Sue
Last Name (key): Jones
Put: Sue Jones


HashTableTest
Add
Get
Remove
Empty
Contains Key
Clear Table
List Objects
List Keys

First Name: Bob
Last Name (key): Smith
Put: Bob Smith

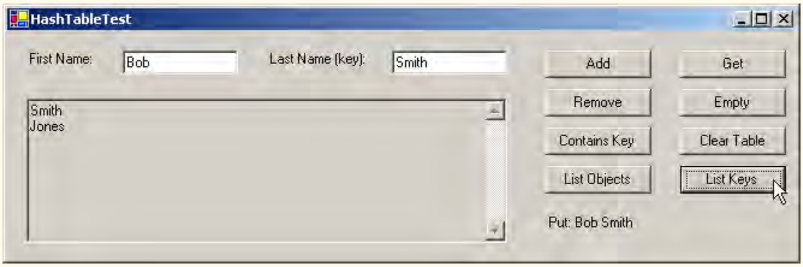

HashTableTest
Add
Get
Remove
Empty
Contains Key
Clear Table
List Objects
List Keys

First Name: Bob
Last Name (key): Smith

Bob Smith
Sue Jones

Put: Bob Smith

Fig. 23.26 Using the `Hashtable` class. (Part 4 of 5.)



First Name: Bob

Last Name (key): Smith

Smith
Jones

Put: Bob Smith

Fig. 23.26 Using the **`Hashtable`** class. (Part 5 of 5.)

Event handler **`addButton_Click`** (lines 57–75) reads the first name and last name of an employee from the user interface, creates an object of class Employee (defined at lines 153–170) and adds that **`Employee`** to the **`Hashtable`** with method ***`Add`*** (line 66). This method receives two arguments—a key object, and a value object. In this example, the key is the last name of the **`Employee`** (a **`string`**), and the value is the corresponding **`Employee`** object. An **`ArgumentException`** occurs if the **`Hashtable`** already contains the key or if the key is **`null`**.

Event handler **`getButton_Click`** (lines 78–88) retrieves the object associated with a specific key, using the **`Hashtable`**'s subscript operator as shown on line 81. The expression in square brackets is the key for which the **`Hashtable`** should return the corresponding object. If the key is not found, the result is **`null`**.

Event handler **`removeButton_Click`** (lines 91–96) invokes **`Hashtable`** method ***`Remove`*** to delete a key and its associated object from the **`Hashtable`**. If the key does not exist in the table, nothing happens.

Event handler **`emptyButton_Click`** (lines 99–104) uses **`Hashtable`** property ***`Count`*** to determine whether the **`Hashtable`** is empty (i.e., **`Count`** is 0).

Event handler **`containsKeyButton_Click`** (lines 107–112) invokes **`Hashtable`** method ***`ContainsKey`*** to determine whether the **`Hashtable`** contains the specified key. If so, the method returns **`true`**; otherwise, it returns **`false`**.

Event handler **`clearTableButton_Click`** (lines 115–120) invokes **`Hashtable`** method ***`Clear`*** to delete all **`Hashtable`** entries.

Class **`Hashtable`** provides method **`GetEnumerator`** that returns an enumerator of type ***`IDictionaryEnumerator`***, which derives from **`IEnumerator`**. Such enumerators provide properties ***`Key`*** and ***`Value`*** to access the information for a key/value pair. The event handler at lines 123–134 (**`listObjectsButton_click`**) uses the **`Value`** property of the enumerator to output the objects in the **`Hashtable`**. The event handler at lines 123–134 (**`listKeysButton_click`**) uses the **`Key`** property of the enumerator to output the keys in the **`Hashtable`**.

SUMMARY

- Dynamic data structures can grow and shrink at execution time.
- Creating and maintaining dynamic data structures requires dynamic memory allocation—the ability for a program to obtain more memory at execution time (to hold new nodes) and to release memory no longer needed.

- The limit for dynamic memory allocation can be as large as the available physical memory in the computer or the amount of available disk space in a virtual-memory system.
- Operator **`new`** takes as an operand the type of the object being dynamically allocated and returns a reference to a newly created object of that type. If no memory is available, **`new`** throws an **`OutOfMemoryException`**.
- A self-referential class contains a data member that refers to an object of the same class type. Self-referential objects can be linked to form useful data structures such as lists, queues, stacks and trees.
- A linked list is a linear collection (i.e., a sequence) of self-referential class objects called nodes, connected by reference links.
- A node can contain data of any type, including objects of other classes.
- A linked list is accessed via a reference to the first node of the list. Each subsequent node is accessed via the link-reference member stored in the previous node.
- By convention, the link reference in the last node of a list is set to **`null`** to mark the end of the list.
- Stacks are important in compilers and operating systems.
- A stack is a constrained version of a linked list—new nodes can be added to a stack and removed from a stack only at the top. A stack is referred to as a last-in, first-out (LIFO) data structure.
- The primary stack operations are push and pop. Operation push adds a new node to the top of the stack. Operation pop removes a node from the top of the stack and returns the data object from the popped node.
- Queues represent waiting lines. Insertions occur at the back (also referred to as the tail) of a queue and deletions occur from the front (also referred to as the head) of a queue.
- A queue is similar to a checkout line in a supermarket: The first person in line is served first; other customers enter the line only at the end and wait to be served.
- Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue. For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.
- The insert and remove operations for a queue are known as enqueue and dequeue.
- Binary trees facilitate high-speed searching and sorting of data.
- Tree nodes contain two or more links.
- A binary tree is a tree whose nodes all contain two links. The root node is the first node in a tree.
- Each link in the root node refers to a child. The left child is the first node in the left subtree and the right child is the first node in the right subtree.
- The children of a node are called siblings. A node with no children is called a leaf node.
- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the values that subtree's parent node and the values in any right subtree are greater than the values in that subtree's parent node.
- A node can be inserted only as a leaf node in a binary search tree.
- An inorder traversal of a binary search tree processes the node values in ascending order.
- The process of creating a binary search tree actually sorts the data—hence, the term "binary tree sort."
- In a preorder traversal, the value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed, then the values in the right subtree are processed.

- In a postorder traversal, the value in each node is processed after the node's left and right subtrees are processed.
- The binary search tree facilitates duplicate elimination. As the tree is created, attempts to insert a duplicate value are recognized because a duplicate follows the same "go left" or "go right" decisions on each comparison as the original value did. Thus, the duplicate eventually is compared with a node containing the same value. The duplicate value may simply be discarded at this point.
- Class **`ArrayList`** can be used as a dynamically growing array.
- **`ArrayList`** method **`Add`** adds an **`object`** to the **`ArrayList`**.
- **`ArrayList`** method **`Remove`** removes the first occurrence of the specified **`object`** from the **`ArrayList`**.
- The **`ArrayList`** subscript operator accesses elements of an **`ArrayList`** as if it were an array.
- Class **`Stack`** is provided in the **`System.Collections`** namespace.
- **`Stack`** method **`Push`** performs the push operation on the **`Stack`**.
- **`Stack`** method **`Pop`** performs the pop operation on the **`Stack`**.
- Class **`Hashtable`** is provided in the **`System.Collections`** namespace.
- **`Hashtable`** method **`Add`** adds a key/value pair to the **`Hashtable`**.
- Any class that implements the **`IEnumerator`** interface must define methods **`MoveNext`** and **`Reset`** and the **`Current`** property.
- Method **`MoveNext`** must be called before the **`Current`** property is accessed for the first time.
- Methods **`MoveNext`** and **`Reset`** throw an **`InvalidOperationException`** if the contents of the collection were modified in any way after the enumerator's creation.

TERMINOLOGY

`Add` method of **`ArrayList`**
`ArgumentException`
`ArrayList` class
binary tree
`BinarySearch` method of **`ArrayList`**
`Capacity` property of **`ArrayList`**
`Clear` method of **`ArrayList`**
`Clear` method of **`Hashtable`**
collection
`Contains` method of **`ArrayList`**
`Contains` method of **`Stack`**
`ContainsKey` method of **`Hashtable`**
`Count` property of **`ArrayList`**
`Count` property of **`Stack`**
`Current` property of **`IEnumerator`**
data structures
dynamic data structures
enumerator
`GetEnumerator` method of **`IEnumerable`**
`GetHashCode` method of **`Object`**
`Hashtable` class
head
`IDictionaryEnumerator` interface
`IEnumerator` interface
`IndexOf` method of **`ArrayList`**
`InvalidOperationException`
linked list
`MoveNext` method of **`IEnumerator`**
`Peek` method of **`Stack`**
`Pop` method of **`Stack`**
`Push` method of **`Stack`**
queue
`Remove` method of **`ArrayList`**
`Remove` method of **`Hashtable`**
`RemoveAt` method of **`ArrayList`**
`RemoveRange` method of **`ArrayList`**
`Reset` method of **`IEnumerator`**
searching
self-referential class
`Sort` method of **`ArrayList`**
sorting
stack
`Stack` class
`System.Collections` namespace
`TrimToSize` method of **`ArrayList`**
waiting line

SELF-REVIEW EXERCISES

23.1 State whether each of the following is *true* or *false*. If *false*, explain why.

a) In a queue, the first item to be added, is the last item to be removed.
b) Trees can have no more than two child nodes per node.
c) A tree node with no children is called a leaf node.
d) Class `Stack` is in the `System.Collections` namespace.
e) A class implementing interface `IEnumerator` must define only methods `MoveNext` and `Reset`.
f) A hashtable stores key/value pairs.
g) Linked list nodes are stored contiguously in memory.
h) The primary operations of the stack data structure are enqueue and dequeue.
i) Lists, stacks and queues are linear data structures.

23.2 Fill in the blanks in each of the following statements:

a) A __________ class is used to define nodes that form dynamic data structures, which can grow and shrink at execution time.
b) Operator __________ allocates memory dynamically; this operator returns a reference to the allocated memory.
c) A __________ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list; this data structure returns node values in last-in, first-out order.
d) A queue is a __________ data structure, because the first nodes inserted are the first nodes removed.
e) A __________ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.
f) A __________ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
g) The nodes of a __________ tree contain two link members.
h) `IEnumerator` method __________ advances the enumerator to the next item.
i) The tree-traversal algorithm that processes the node and then processes all the nodes to its left followed by all the nodes to its right is called __________.
j) If the collection it references was altered after the enumerator's creation, calling method `Reset` will cause an __________.

ANSWERS TO SELF-REVIEW EXERCISES

23.1 a) False. A queue is a first-in, first-out data structure—the first item added is the first item removed. b) False. In general, trees may have as many child nodes per node as is necessary. Only binary trees are restricted to no more than two child nodes per node. c) True. d) True. e) False. The class must also implement property `Current`. f) True. g) False. Linked-list nodes are logically contiguous, but they need not be stored in a physically contiguous memory space. h) False. Those are the primary operations of a queue. The primary operations of a stack are push and pop. i) True.

23.2 a) self-referential. b) `new`. c) stack. d) first-in, first-out (FIFO). e) queue. f) tree. g) binary. h) `MoveNext`. i) preorder. j) `InvalidOperationException`.

EXERCISES

23.3 Write a program that merges two ordered list objects of integers into a single ordered list object of integers. Method `Merge` of class `ListMerge` should receive references to each of the list objects to be merged and should return a reference to the merged list object.

23.4 Write a program that inputs a line of text and uses a stack object to print the line reversed.

23.5 Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

23.6 Stacks are used by compilers to help in the process of evaluating expressions and in generating machine language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like `3 + 4` and `7 / 9`, in which the operator (`+` or `/` here) is written between its operands—this is called *infix notation*. Computers "prefer" *postfix notation*, in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, then evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, and in each algorithm the stack is used for a different purpose.

In this exercise, you will write a C# version of the infix-to-postfix conversion algorithm. In the next exercise, you will write a C# version of the postfix expression evaluation algorithm. In a later exercise, you will discover that code you write in this exercise can help you implement a complete working compiler.

Write class **`InfixToPostfixConverter`** to convert an ordinary infix arithmetic expression (assume a valid expression is entered), with single-digit integers, such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression (note that no parentheses are needed) is

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into **`StringBuilder infix`**, then use class **`StackComposition`** (implemented in Fig. 23.12) to help create the postfix expression in **`StringBuilder postfix`**. The algorithm for creating a postfix expression is as follows:

a) Push a left parenthesis `'('` on the stack.
b) Append a right parenthesis `')'` to the end of **`infix`**.
c) While the stack is not empty, read **`infix`** from left to right and do the following:
    - If the current character in **`infix`** is a digit, append it to **`postfix`**.
    - If the current character in **`infix`** is a left parenthesis, push it onto the stack.
    - If the current character in **`infix`** is an operator:
        - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and append the popped operators to **`postfix`**.
        - Push the current character in **`infix`** onto the stack.
    - If the current character in **`infix`** is a right parenthesis:
        - Pop operators from the top of the stack and append them to **`postfix`** until a left parenthesis is at the top of the stack.
        - Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`%`	modulus

Some of the methods you may want to provide in your program follow:

a) Method **`ConvertToPostfix`**, which converts the infix expression to postfix notation.

b) Method **`IsOperator`**, which determines whether **`c`** is an operator.

c) Method **`Precedence`**, which determines whether the precedence of **`operator1`** (from the infix expression) is less than, equal to or greater than the precedence of **`operator2`** (from the stack). The method returns **`true`** if **`operator1`** has lower precedence than **`operator2`**. Otherwise, **`false`** is returned.

d) Add this method to the class definition for class **`StackComposition`**.

23.7 Write class **`PostfixEvaluator`**, which evaluates a postfix expression (assume it is valid) such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a **`StringBuilder`**. Using class **`StackComposition`** from Exercise 23.6, the program should scan the expression and evaluate it. The algorithm is as follows:

a) Append a right parenthesis (**`')'`**) to the end of the postfix expression. When the right-parenthesis character is encountered, no further processing is necessary.

b) When the right-parenthesis character has not been encountered, read the expression from left to right.

    If the current character is a digit do the following:
        Push its integer value on the stack (the integer value of a digit character is its value in the computer's character set minus the value of **`'0'`** in Unicode).
    Otherwise, if the current character is an *operator*:
        Pop the two top elements of the stack into variables **`x`** and **`y`**.
        Calculate **`y`** *operator* **`x`**.
        Push the result of the calculation onto the stack.

c) When the right parenthesis is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[*Note*: In b) above (based on the sample expression at the beginning of this exercises), if the operator is **`'/'`**, the top of the stack is **`2`** and the next element in the stack is **`8`**, then pop **`2`** into **`x`**, pop **`8`** into **`y`**, evaluate **`8 / 2`** and push the result, **`4`**, back on the stack. This note also applies to operator **`'-'`**.] The arithmetic operations allowed in an expression are:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	exponentiation
`%`	modulus

You may want to provide the following methods:

a) Method **`EvaluatePostfixExpression`**, which evaluates the postfix expression.

b) Method **`Calculate`**, which evaluates the expression **`op1`** *operator* **`op2`**.

23.8 (*Binary Tree Delete*) In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. There are three cases that are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child or the item is contained in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the reference in the parent node is set to null.

If the item to be deleted is contained in a node with one child, the reference in the parent node is set to reference the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the reference in the parent node simply cannot be assigned to reference one of the children of the node to be deleted. In most cases, the resulting binary search tree would not adhere to the following characteristic of binary search trees (with no duplicate values): *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.*

Which node is used as a *replacement node* to maintain this characteristic—either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let us consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the reference to the right child of the current node is null. We are now referencing the replacement node which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node to null.
d) Set the reference to the right subtree in the replacement node to reference the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node's position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

a) Store the reference to the node to be deleted in a temporary reference variable.
b) Set the reference in the parent of the node being deleted to reference the replacement node.
c) Set the reference in the parent of the replacement node reference to the left child of the replacement node.
d) Set the reference to the right subtree in the replacement node reference to the right subtree of the node to be deleted.
e) Set the reference to the left subtree in the replacement node to reference the left subtree of the node to be deleted.

Write method `DeleteNode`, which takes as its argument the value to be deleted. Method `DeleteNode` should locate in the tree the node containing the value to be deleted and use the algorithms discussed here to delete the node. If the value is not found in the tree, the method should print a message that indicates whether the value is deleted. Modify the program of Fig. 23.18 to use this method. After deleting an item, call the methods `InorderTraversal`, `PreorderTraversal` and `PostorderTraversal` to confirm that the delete operation was performed correctly.

23.9 (*Level-Order Binary Tree Traversal*) The program of Fig. 23.18 illustrated three recursive methods of traversing a binary tree—inorder, preorder, and postorder traversals. This exercise presents the *level-order traversal* of a binary tree, in which the node values are printed level by level, starting at the root-node level. The nodes on each level are printed from left to right. The level-order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

a) Insert the root node in the queue.
b) While there are nodes left in the queue, do the following:
    Get the next node in the queue.
    Print the node’s value.
    If the reference to the left child of the node is not null:
        Insert the left child node in the queue.
    If the reference to the right child of the node is not null:
        Insert the right child node in the queue.

Write method **`LevelOrder`** to perform a level-order traversal of a binary tree object. Modify the program of Fig. 23.18 to use this method. [*Note*: You also will need to use the queue-processing methods of Fig. 23.13 in this program.]

24

Accessibility

Objectives

- To introduce the World Wide Web Consortium's Web Content Accessibility Guidelines 1.0 (WCAG 1.0).
- To understand how to use the `alt` attribute of the HTML `<img>` tag to describe images to people with visual impairments, mobile-Web-device users and others unable to view images.
- To understand how to make tables more accessible to page readers.
- To understand how to verify that XHTML tags are used properly and to ensure that Web pages can be viewed on any type of display or reader.
- To understand how VoiceXML™ and CallXML™ are changing the way in which people with disabilities access information on the Web.
- To introduce the various accessibility aids offered in Windows 2000.

'Tis the good reader that makes the good book...
Ralph Waldo Emerson

I once was lost, but now am found,
Was blind, but now I see.
John Newton

Outline

24.1 Introduction

24.2 Regulations and Resources

24.3 Web Accessibility Initiative

24.4 Providing Alternatives for Images

24.5 Maximizing Readability by Focusing on Structure

24.6 Accessibility in Visual Studio .NET

  24.6.1 Enlarging Toolbar Icons

  24.6.2 Enlarging the Text

  24.6.3 Modifying the Toolbox

  24.6.4 Modifying the Keyboard

  24.6.5 Rearranging Windows

24.7 Accessibility in C#

24.8 Accessibility in XHTML Tables

24.9 Accessibility in XHTML Frames

24.10 Accessibility in XML

24.11 Using Voice Synthesis and Recognition with VoiceXML™

24.12 CallXML™

24.13 JAWS® for Windows

24.14 Other Accessibility Tools

24.15 Accessibility in Microsoft® Windows® 2000

  24.15.1 Tools for People with Visual Impairments

  24.15.2 Tools for People with Hearing Impairments

  24.15.3 Tools for Users Who Have Difficulty Using the Keyboard

  24.15.4 Microsoft Narrator

  24.15.5 Microsoft On-Screen Keyboard

  24.15.6 Accessibility Features in Microsoft Internet Explorer 5.5

24.16 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

24.1 Introduction

Throughout this book, we discuss the creation of C# applications. Later chapters also introduce the development of Web-based content using Web Forms, ASP .NET, XHTML and XML. In this chapter, we explore the topic of *accessibility*, which refers to the level of usability that an application or Web site provides to people with various disabilities. Disabilities that might affect an individual's computer or Internet usage are common; they include visual impairments, hearing impairments, other physical injuries (such as the inability to use a keyboard or a mouse) and learning disabilities. In today's computing environment, such impediments prevent many users from taking full advantage of applications and Web content.

The design of applications and sites to meet the needs of individuals with disabilities should be a priority for all software companies and e-businesses. People affected by disabilities represent a significant portion of the population, and legal ramifications could exist for companies that discriminate by failing to provide adequate and universal access to their resources. In this chapter, we explore the World Wide Web Consortium's *Web Accessibility Initiative* and its guidelines and review various laws regarding the availability of computing and Internet resources to people with disabilities. We also highlight companies that have developed systems, products and services that meet the needs of this demographic. As students use C# and its related technologies to design applications and Web sites, they should keep in mind the accessibility requirements and recommendations that we discuss in this chapter.

24.2 Regulations and Resources

Over the past several years, the United States has taken legislative steps to ensure that people with disabilities are given the tools they need to use computers and access the Web. A wide variety of legislation, including the *Americans With Disabilities Act* (ADA) of 1990, governs the provision of computer and Web accessibility (Fig. 24.1). These laws have inspired significant legal action. For example, according to the ADA, companies are required to offer equal access to individuals with visual problems. The National Federation for the Blind (NFB) cited this law in a 1999 suit against AOL, responding to the company's failure to make its services available to individuals with disabilities.

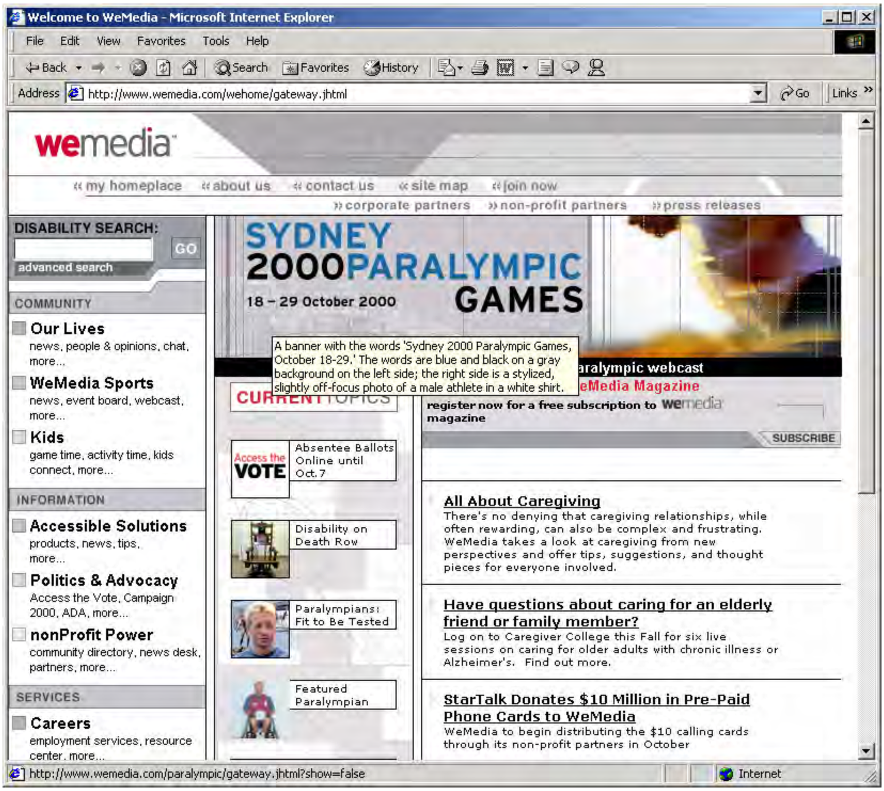
There are 54 million Americans with disabilities, and these individuals represent an estimated $1 trillion in annual purchasing power. In addition to legislation, many organizations and resources focus on assisting individuals with disabilities to access computers and the Internet. **`WeMedia.com`**™ (Fig. 24.2) is a Web site that provides news, information, products and services to the millions of people with disabilities and to their families, friends and caregivers.

Act	**Purpose**
Americans with Disabilities Act	The ADA prohibits discrimination on the basis of disability in employment, state and local government, public accommodations, commercial facilities, transportation and telecommunications.
Telecommunications Act of 1996	The Telecommunications Act of 1996 contains two amendments to Section 255 and Section 251(a)(2) of the Communications Act of 1934. These amendments require that communication devices, such as cell phones, telephones and pagers, be accessible to individuals with disabilities.

Fig. 24.1 Acts designed to improve Internet and computer accessibility for people with disabilities. (Part 1 of 2.)

Act	Purpose
Individuals with Disabilities Education Act of 1997	The Individuals with Disabilities Education Act stipulates that education materials in schools must be made accessible to children with disabilities.
Rehabilitation Act	Section 504 of the Rehabilitation Act states that college sponsored activities receiving federal funding cannot discriminate against individuals with disabilities. Section 508 mandates that all government institutions receiving federal funding must design their Web sites so that they are accessible to individuals with disabilities. Businesses that sell services to the government also must abide by this act.

Fig. 24.1 Acts designed to improve Internet and computer accessibility for people with disabilities. (Part 2 of 2.)


Welcome to WeMedia - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
wemedia
my homeplace
about us
contact us
site map
join now
corporate partners
non-profit partners
press releases
DISABILITY SEARCH:
GO
advanced search
COMMUNITY
Our Lives
news, people & opinions, chat, more...
WeMedia Sports
news, event board, webcast, more...
Kids
game time, activity time, kids connect, more...
INFORMATION
Accessible Solutions
products, news, tips, more...
Politics & Advocacy
Access the Vote, Campaign 2000, ADA, more...
nonProfit Power
community directory, news desk, partners, more...
SERVICES
Careers
employment services, resource center, more...
SYDNEY 2000 PARALYMPIC GAMES
18 – 29 October 2000
aralympic webcast
eMedia Magazine
CURRENT TOPICS
Access the VOTE
Absentee Ballots Online until Oct. 7
Disability on Death Row
Paralympians: Fit to Be Tested
Featured Paralympian
SUBSCRIBE
Internet

http://www.wemedia.com/wehome/gateway.jhtml

A banner with the words 'Sydney 2000 Paralympic Games, October 18-29.' The words are blue and black on a gray background on the left side; the right side is a stylized, slightly off-focus photo of a male athlete in a white shirt.

register now for a free subscription to wemedia magazine

All About Caregiving
There's no denying that caregiving relationships, while often rewarding, can also be complex and frustrating. WeMedia takes a look at caregiving from new perspectives and offer tips, suggestions, and thought pieces for everyone involved.

Have questions about caring for an elderly friend or family member?
Log on to Caregiver College this Fall for six live sessions on caring for older adults with chronic illness or Alzheimer's. Find out more.

StarTalk Donates $10 Million in Pre-Paid Phone Cards to WeMedia
WeMedia to begin distributing the $10 calling cards through its non-profit partners in October

http://www.wemedia.com/paralympic/gateway.jhtml?show=false

Fig. 24.2 We Media's home page. (Courtesy of WeMedia, Inc.)

As these laws and resources exemplify, computer and Internet accessibility for individuals with disabilities is quickly becoming a reality. Such accessibility enables individuals with disabilities to work in a vast array of new fields. This is partly because the Internet provides a medium through which disabled people can telecommute to jobs and interact easily with others without traveling. Such technologies as voice activation, visual enhancers and auditory aids create additional employment opportunities. For example, people with visual impairments can use computer monitors with enlarged text, and people with physical impairments can use head pointers with on-screen keyboards. In the remaining sections of this chapter, we explore various organizations, techniques, products and services that help provide computer and Internet access to people with disabilities.

24.3 Web Accessibility Initiative

Currently, most Web sites are considered to be either partially or totally inaccessible to people with visual, learning or mobility impairments. Total accessibility is difficult to achieve, because of the variety of disabilities that must be accommodated and because of problems resulting from language barriers and hardware and software inconsistencies. However, a high level of accessibility is attainable. As more people with disabilities begin to use the Internet, it is imperative that Web-site designers increase the accessibility of their sites. Although computer and Web accessibility is the focus of some recent legislation, standards organizations also see the need for industry recommendations. In an attempt to address issues of accessibility, the World Wide Web Consortium (W3C) launched the *Web Accessibility Initiative* (WAI™) in April 1997. To learn more about the WAI or to read its mission statement, visit **`www.w3.org/WAI`**.

This chapter explains various techniques used to develop accessible Web sites. In 1999, the WAI published the *Web Content Accessibility Guidelines* (*WCAG*) *1.0* to help businesses determine whether their Web sites are universally accessible. The WCAG 1.0 (available at **`www.w3.org/TR/WCAG10`**) uses checkpoints to list specific accessibility requirements. Each checkpoint is accompanied by a corresponding priority rating that indicates the requirement's level of importance. *Priority-one checkpoints* are goals that must be met to ensure accessibility; we focus on these points in this chapter. *Priority-two checkpoints*, though not essential, are highly recommended. If these checkpoints are not satisfied, people with certain disabilities will experience difficulty accessing Web sites. *Priority-three checkpoints* slightly improve accessibility.

At the time of publication, the WAI was working on *WCAG 2.0*; a working draft of this publication can be found at **`www.w3.org/TR/WCAG20`**. A single checkpoint in the WCAG 2.0 Working Draft might encompass several checkpoints from WCAG 1.0. Once WCAG 2.0 has been reviewed and published by the W3C, its checkpoints will supersede those of WCAG 1.0. Furthermore, the new version can be applied to a wider range of markup languages (i.e., XML, WML, etc.) and content types than can its predecessor.

The WAI also presents a supplemental checklist of *quick tips*, which reinforce ten important points relating to accessible Web–site design. More information on the WAI Quick Tips can be found at **`www.w3.org/WAI/References/Quicktips`**.

24.4 Providing Alternatives for Images

One important WAI requirement specifies that every image on a Web page should be accompanied by a textual description that clearly defines the purpose of the image. To accomplish this task, Web developers can use the **`alt`** attribute of the **`img`** and **`input`** tags to include a textual equivalent for every image or graphic included on a site.

Web developers who do not use the **`alt`** attribute to provide text equivalents increase the difficulties that people with visual impairments experience in navigating the Web. Specialized *user agent*s (or *accessibility aids*), such as *screen readers* (programs that allow users to hear all text that is displayed on their screens) and *braille displays* (devices that receive data from screen-reading software and then output the data as braille), enable people with visual impairments to access text-based information that normally is displayed on the screen. A user agent visually interprets Web-page source code and translates it into a format that is accessible to people with various disabilities. Web browsers, such as Microsoft Internet Explorer and Netscape Communicator, and the screen readers mentioned throughout this chapter are examples of user agents.

Similarly, Web pages that do not provide text equivalents for video and audio clips are difficult for people with visual and hearing impairments to access. Screen readers cannot interpret images, movies and most other non-XHTML content from these Web pages. However, by providing multimedia-based information in a variety of ways (e.g., using the **`alt`** attribute or providing in-line descriptions of images), Web designers can help maximize the accessibility of their sites' content.

Web designers should provide useful and appropriate text equivalents in the **`alt`** attribute for use by nonvisual user agents. For example, if the **`alt`** attribute describes a sales-growth chart, it should provide a brief summary of the data, but should not describe the data in the chart. Instead, a complete description of the chart's data should be included in the ***`longdesc`*** (long description) *attribute*, which is intended to augment the **`alt`** attribute's description. The **`longdesc`** attribute contains a link to a Web page describing the image or multimedia content. Currently, most Web browsers do not support the **`longdesc`** attribute. An alternative to the **`longdesc`** attribute is *D-link*, which provides descriptive text about graphs and charts. More information on D-links can be obtained at the *CORDA Technologies* Web site (**`www.corda.com`**).

The use of a screen reader to facilitate Web-site navigation can be time-consuming and frustrating, because screen readers cannot interpret pictures and other graphical content. The inclusion of a link at the top of each Web page providing direct access to the page's content could allow disabled users to bypass long lists of navigation links and other irrelevant or inaccessible content. This jump can save time and eliminate frustration for individuals with visual impairments.

Emacspeak (**`www.cs.cornell.edu/home/raman/emacspeak/emacspeak.html`**) is a screen interface that improves the quality of Internet access for individuals with visual disabilities by translating text to voice data. The open-source product also implements auditory icons that play various sounds. Emacspeak can be customized with Linux operating systems and provides support for the IBM *ViaVoice* speech engine.

In March 2001, We Media introduced another user agent, the *WeMedia Browser*, which allows people with vision impairments and cognitive disabilities (such as dyslexia) to use the Internet more conveniently. The WeMedia Browser enhances traditional browser capabilities by providing oversized buttons and keystroke commands that assist in navigation. The browser "reads" text that the user selects, allowing the user to control the speed and volume at which the browser reads the contents of the Web page. The WeMedia Browser free download is available at **`www.wemedia.com`**.

IBM Home Page Reader (HPR) is another browser that “reads” text selected by the user. The HPR uses IBM ViaVoice technology to synthesize an audible voice. A trial version of HPR is available at **`www-3.ibm.com/able/hpr.html`**.

24.5 Maximizing Readability by Focusing on Structure

Many Web sites use XHTML tags for aesthetic purposes, ignoring the tags’ intended functions. For example, the **`<h1>`** heading tag often is used erroneously to make text large and bold, rather than to indicate a major section head for content. This practice might create a desired visual effect, but it causes problems for screen readers. When the screen-reader software encounters the **`<h1>`** tag, it might verbally inform the user that a new section has been reached. If this is not in fact the case, the **`<h1>`** tag might confuse users. Therefore, developers should use the **`h1`** only in accordance with its XHTML specifications (e.g., to mark up a heading that introduces an important section of a document). Instead of using **`h1`** to make text large and bold, developers can use CSS (Cascading Style Sheets) or XSL (Extensible Stylesheet Language) to format and style the text. For further examples of this nature, refer to the WCAG 1.0 Web site at **`www.w3.org/TR/WCAG10`**. [*Note:* The **`<strong>`** tag also can be used to make text bold; however, screen readers emphasize bold text, which affects the inflection of what is spoken.]

Another accessibility issue is *readability*. When creating a Web page intended for the general public, it is important to consider the reading level (i.e., level of difficulty to read and understand) at which content is written. Web-site designers can make their sites easier to read by using shorter words. Furthermore, slang terms and other nontraditional language could be problematic for users from other countries, so developers should limit the use of such words.

WCAG 1.0 suggests using a paragraph’s first sentence to convey its subject. When a Web site states the point of a paragraph in this paragraph’s first sentence, it is easier for individuals with disabilities both to find crucial information and to bypass unwanted material.

The *Gunning Fog Index*, a formula that produces a readability grade when applied to a text sample, can evaluate a Web site’s readability. To obtain more information about the Gunning Fog Index, visit **`www.trainingpost.org/3-2-inst.htm`**.

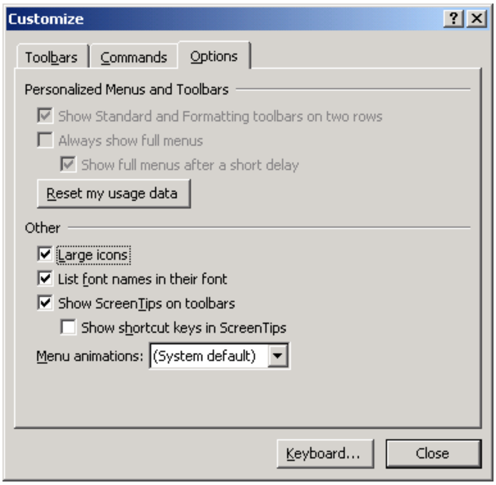
24.6 Accessibility in Visual Studio .NET

In the previous sections, we have outlined various accessibility guidelines presented in the W3C’s Web Accessibility initiative. However, Visual Studio .NET provides its own guidelines for designing accessible software within its programming environment. For instance, one guideline recommends reserving the use of color for the enhancement or emphasis of information, instead of for aesthetic purposes. A second guideline recommends providing information about objects (e.g., desktop icons and open windows) to the accessibility aids (specialized software that renders applications to individuals with disabilities). Such information might include the name, location and size of a window. A third guideline recommends designing user interfaces so that they can accommodate user preferences. For example, people with visual disabilities should be able to modify the font size of a user interface. A fourth guideline recommends allowing users to adjust the time setting for applications that have time constraints. For example, users with mobility or speech disabilities might experience difficulty when using applications that require users to enter input within a predetermined period of time (such as 10 seconds). However, if such applications provide adjustable time settings, users can modify the settings to suit their needs.

In addition to suggesting guidelines the help developers create accessible applications, Visual Studio .NET also offers features that enable disabled individuals to use the development environment itself. For example, users can enlarge icons and text, customize the toolbox and keyboard and rearrange windows. The next subsections illustrate these capabilities.

24.6.1 Enlarging Toolbar Icons

To enlarge icons in Visual Studio, select **Customize** from the **Tools** menu. In the **Customize** window's **Options** tab, select the **Large Icons** check box (Fig. 24.3), and select **Close**. Figure 24.4 depicts the enlarged icons on the Visual Studio development window.


Customize
Toolbars
Commands
Options
Reset my usage data
Keyboard...
Close

Personalized Menus and Toolbars

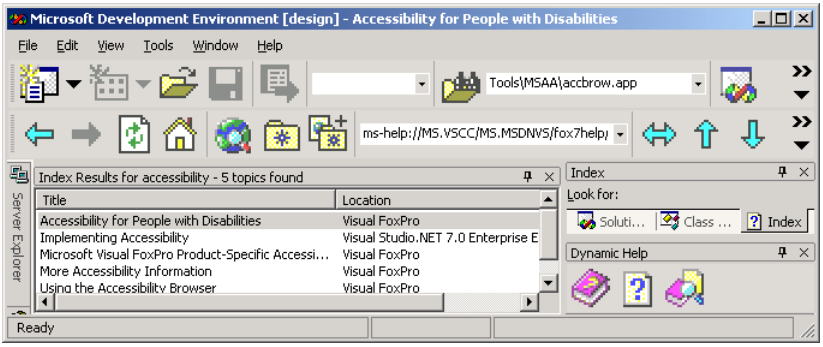
☑ Show Standard and Formatting toolbars on two rows
☐ Always show full menus
☑ Show full menus after a short delay

Other

☑ Large icons
☑ List font names in their font
☑ Show ScreenTips on toolbars
☐ Show shortcut keys in ScreenTips

Menu animations: (System default)

Fig. 24.3 Enlarging icons using the **Customize** feature.


Microsoft Development Environment [design] - Accessibility for People with Disabilities
File
Edit
View
Tools
Window
Help
Tools\MSAA\accbrow.app
ms-help://MS.VSCC/MS.MSDNVS/fox7help/
Server Explorer
Index
Look for:
Soluti...
Class ...
Index
Dynamic Help
Ready

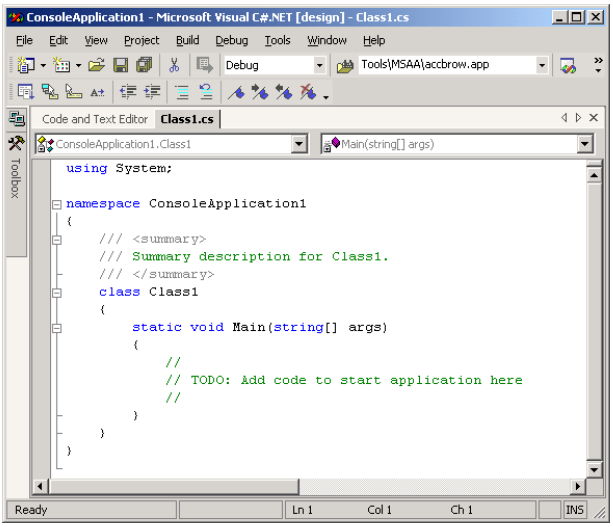
Index Results for accessibility - 5 topics found

Title	Location
Accessibility for People with Disabilities	Visual FoxPro
Implementing Accessibility	Visual Studio.NET 7.0 Enterprise E
Microsoft Visual FoxPro Product-Specific Accessi...	Visual FoxPro
More Accessibility Information	Visual FoxPro
Using the Accessibility Browser	Visual FoxPro

Fig. 24.4 Enlarged icons in the development window.

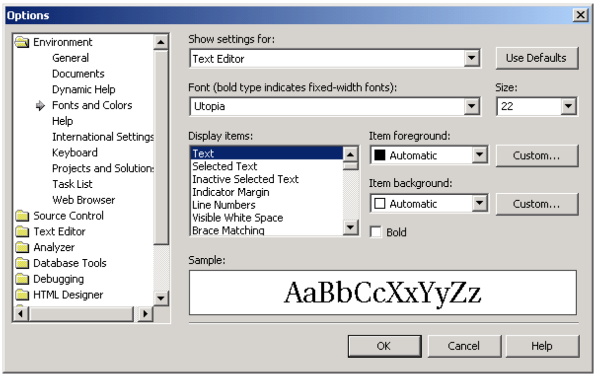
24.6.2 Enlarging the Text

Visual Studio uses the default operating-system font settings when displaying text. However, some individuals cannot read these default font settings, causing the applications to be inaccessible to them. To remedy this, Visual Studio allows users to modify the font size. Select **Options** from the **Tools** menu. In the **Options** window, open the **Environment** directory and choose **Fonts and Colors**. In the **Show settings for** drop-down box, select **Text Editor**. In the **Font** drop-down box, select a different style of font and, in the **Size** drop-down box, select a different font size. Figure 24.5 depicts the **Text Editor** before we modified the font size, Fig. 24.6 shows the **Options** window with new font settings and Fig. 24.7 displays the **Text Editor** after the changes have been applied.


ConsoleApplication1 - Microsoft Visual C#.NET [design] - Class1.cs
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug
Tools\MSAA\accbrow.app
Toolbox
Code and Text Editor
Class1.cs
ConsoleApplication1.Class1
Main(string[] args)
Ready
Ln 1
Col 1
Ch 1
INS

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
        }
    }
}
```

Fig. 24.5 Text Editor before modifying the font size.


Options
Environment
General
Documents
Dynamic Help
Fonts and Colors
Help
International Settings
Keyboard
Projects and Solution:
Task List
Web Browser
Source Control
Text Editor
Analyzer
Database Tools
Debugging
HTML Designer
Use Defaults
Display items:
Text
Selected Text
Inactive Selected Text
Indicator Margin
Line Numbers
Visible White Space
Brace Matching
Custom...
Custom...
OK
Cancel
Help

Show settings for: Text Editor

Font (bold type indicates fixed-width fonts): Utopia

Size: 22

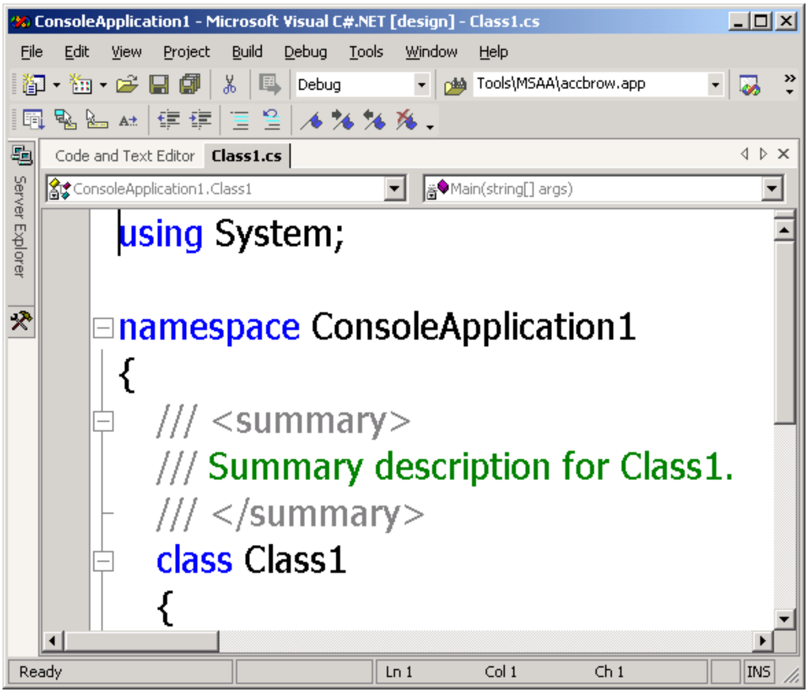
Item foreground: ■ Automatic

Item background: ☐ Automatic

☐ Bold

Sample: AaBbCcXxYyZz

Fig. 24.6 Enlarging text in the **Options** window.


ConsoleApplication1 - Microsoft Visual C#.NET [design] - Class1.cs
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug
Tools\MSAA\accbrow.app
Server Explorer
Code and Text Editor
Class1.cs
ConsoleApplication1.Class1
Main(string[] args)
Ready
Ln 1
Col 1
Ch 1
INS

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
```

Fig. 24.7 Text Editor after the font size is modified.

24.6.3 Modifying the Toolbox

The **Toolbox** feature of Visual Studio contains numerous design elements that facilitate the creation Web applications; however, some developers might use only a few of these design elements. To accommodate the needs of individual developers, Visual Studio allows programmers to customize the toolbox by creating new tabs and then inserting design elements into the tabs. This eliminates the need for users with disabilities to navigate among multiple tabs or scroll through long lists in search of design elements. To create a new tab, right-click any existing tab and select **Add Tab** from the context menu. In the text box, type an identifier for the tab (such as “Frequently Used”) and click *Enter*. By default, the **Pointer** element is placed in all tabs (Fig. 24.8). The **Pointer** element simply allows the cursor to function normally.

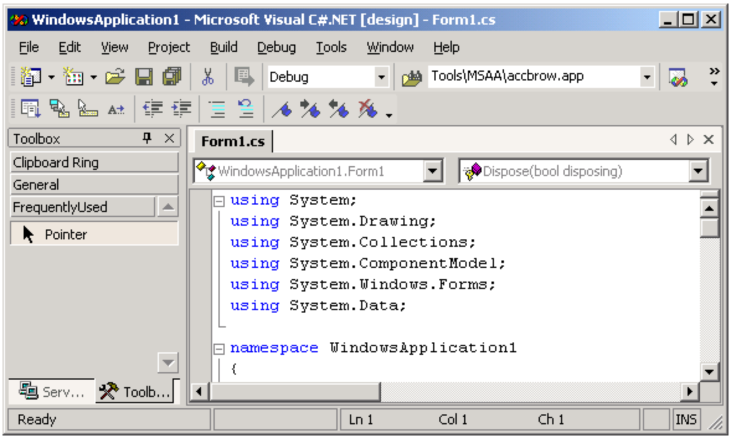
To insert elements into the newly created tab, select **Customize Toolbox** from the **Tools** menu. In the **.NET Framework Components** tab, select the elements to include in the new tab and click **OK**. The selected elements now will appear in the tab.

24.6.4 Modifying the Keyboard

Another accessibility feature in Visual Studio .NET allows individuals with disabilities to customize their keyboards by creating *shortcut keys* (i.e., combinations of keyboard keys that, when pressed together, perform frequent tasks; for example, *Ctrl* + *V* causes text to be pasted from the clipboard). To create a shortcut key, begin by selecting **Options** from the **Tools** menu. In the **Options** window, select the **Keyboard** item from the **Environment** directory. From the **Keyboard mapping scheme** drop-down list, select a scheme and click the **Save As** button. Then, assign a name to the scheme in the **Save Scheme** dialog box and click **OK**. Enter the task of the shortcut key in the **Show commands containing** text box. For example, if we were creating a shortcut key for the paste function, we would enter **Paste** in the text box, or we would select the proper task from the selection list directly below the text box. Then, in the **Use new shortcut** drop-down list, select the applications that will use the shortcut key. If the shortcut key will be used in all applications, select **Global**. Finally, in the **Press shortcut key(s)** text box, assign a shortcut key to the task in the form *non-text key* + *text key*. Valid non-text keys include *Ctrl*, *Shift* and *Alt*; valid text keys include A–Z, inclusive. [*Note*: To enter a non-text key, select the key itself—do not type the word *Ctrl*, *Shift* or *Alt*. It is possible to include more than one non-text key as part of a shortcut key. Do not enter the + symbol.] Thus, a valid shortcut key might be *Ctrl*+*Alt*+*D*. After assigning a shortcut key, select **Assign** and then **OK**. Figure 24.9 illustrates the process of creating a shortcut key for the `NewBreakpoint` function. The shortcut key (*Ctrl*+*Alt*+*D*) is valid only in the **Text Editor**.

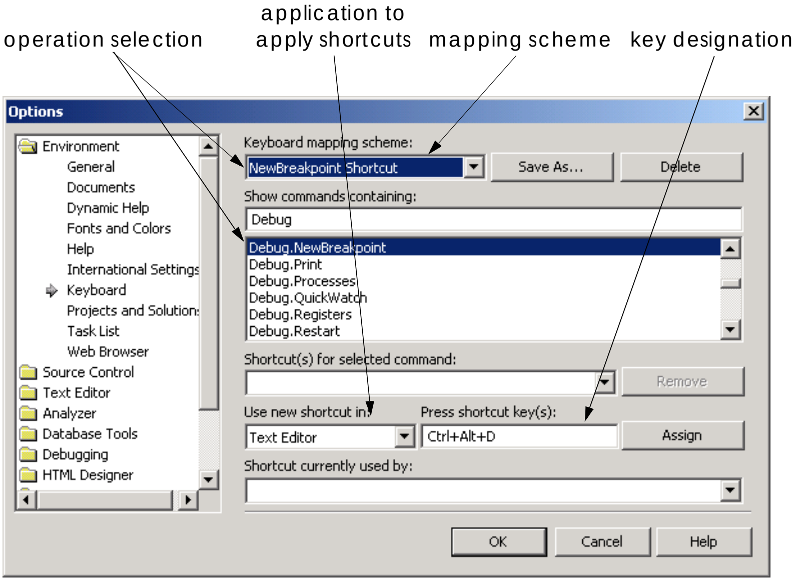
24.6.5 Rearranging Windows

Some screen readers have difficulty interpreting user interfaces that include multiple tabs; this is because most screen readers can read information on only one screen. To accommodate such screen readers, Visual Studio allows developers to customize their user interfaces so that only the console window appears. To remove tabs, select **Options** from the **Tools** menu. Then, in the **Options** window, select the **General** item from the **Environment** directory. In the **Settings** section, select the **MDI environment** radio button and click **OK**. Figure 24.10 depicts the **Options** window, and Fig. 24.11 illustrates a console window with and without tabs.


WindowsApplication1 - Microsoft Visual C#.NET [design] - Form1.cs
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug
Tools\MSAA\accbrow.app
Toolbox
Clipboard Ring
General
FrequentlyUsed
Pointer
Serv...
Toolb...
Form1.cs
WindowsApplication1.Form1
Dispose(bool disposing)
Ready
Ln 1
Col 1
Ch 1
INS

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WindowsApplication1
{
```

Fig. 24.8 Adding tabs to the **Toolbox**.


operation selection
application to apply shortcuts
mapping scheme
key designation
Options
Environment
General
Documents
Dynamic Help
Fonts and Colors
Help
International Settings
Keyboard
Projects and Solutions
Task List
Web Browser
Source Control
Text Editor
Analyzer
Database Tools
Debugging
HTML Designer
Save As...
Delete
Remove
Assign
OK
Cancel
Help

Keyboard mapping scheme: NewBreakpoint Shortcut

Show commands containing: Debug

Debug.NewBreakpoint
Debug.Print
Debug.Processes
Debug.QuickWatch
Debug.Registers
Debug.Restart

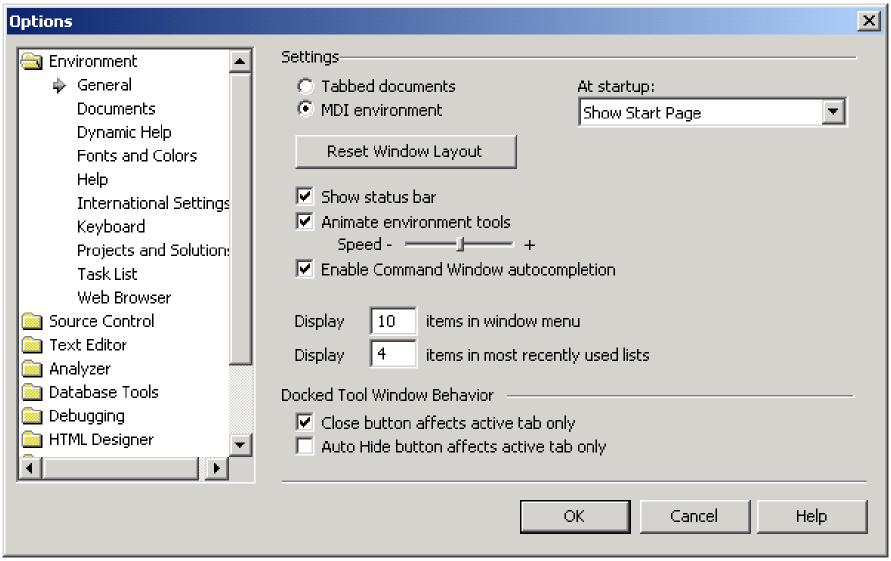
Shortcut(s) for selected command:

Use new shortcut in: Text Editor

Press shortcut key(s): Ctrl+Alt+D

Shortcut currently used by:

Fig. 24.9 Shortcut key creation.


Options
Environment
General
Documents
Dynamic Help
Fonts and Colors
Help
International Settings
Keyboard
Projects and Solutions
Task List
Web Browser
Source Control
Text Editor
Analyzer
Database Tools
Debugging
HTML Designer
Reset Window Layout
OK
Cancel
Help

Settings

○ Tabbed documents
● MDI environment

At startup: Show Start Page

☑ Show status bar
☑ Animate environment tools
Speed - +
☑ Enable Command Window autocompletion

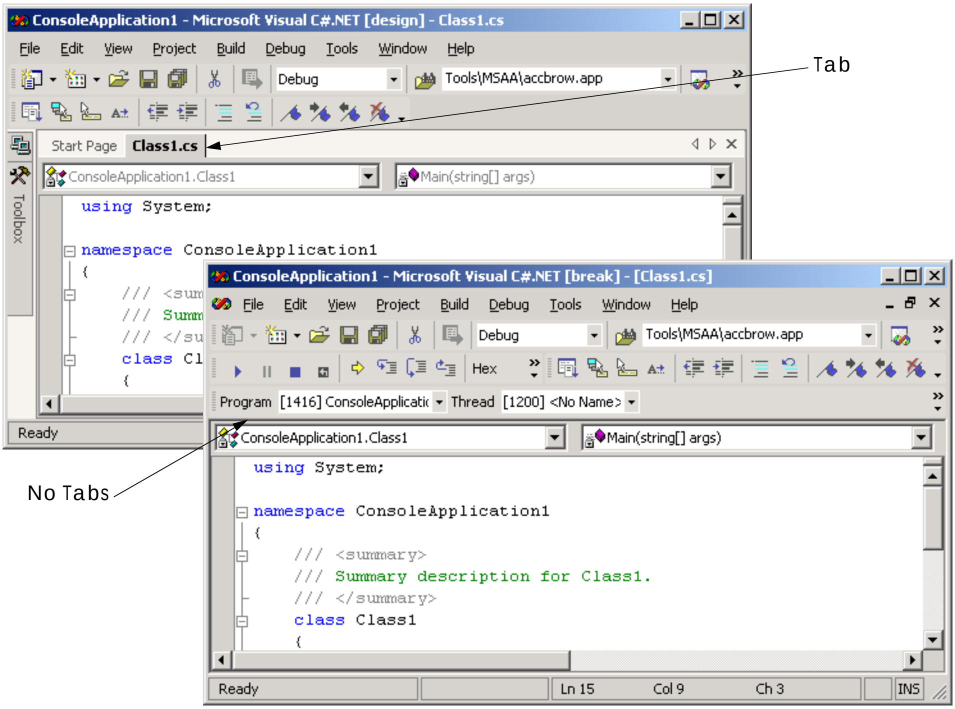
Display 10 items in window menu

Display 4 items in most recently used lists

Docked Tool Window Behavior

☑ Close button affects active tab only
☐ Auto Hide button affects active tab only

Fig. 24.10 Removing tabs from Visual Studio environment.


ConsoleApplication1 - Microsoft Visual C#.NET [design] - Class1.cs
File Edit View Project Build Debug Tools Window Help
Debug
Tools\MSAA\accbrow.app
Start Page
Class1.cs
Toolbox
ConsoleApplication1.Class1
Main(string[] args)
Ready
ConsoleApplication1 - Microsoft Visual C#.NET [break] - [Class1.cs]
File Edit View Project Build Debug Tools Window Help
Debug
Tools\MSAA\accbrow.app
Hex
Program [1416] ConsoleApplicati
Thread [1200] <No Name>
ConsoleApplication1.Class1
Main(string[] args)
Ready
Ln 15
Col 9
Ch 3
INS
Tab
No Tabs

```
using System;

namespace ConsoleApplication1
{
    /// <sum
    /// Summ
    /// </su
    class Cl
    {
```

```
using System;

namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
```

Fig. 24.11 Console windows with tabs and without tabs.

24.7 Accessibility in C#

Visual Studio .NET provides extensive accessibility features and also presents guidelines for creating accessible applications in its development environment. Similar recommendations guide the development of C# applications that are accessible to people with disabilities. It is important that C# programmers gear applications toward as many potential users as possible, rather then toward only the average user. WIth some modifications, most applications can be made accessible to a wide variety of individuals. General guidelines for designing accessible applications are:

1. Use larger-sized fonts—this helps people with visual impairments see the text.
2. Create flexible applications that provide keyboard shortcuts for all features within the application—this allows people to use the application without employing a mouse.
3. Allow information to be conveyed to the user both in a visual and in an audio context.
4. Use graphics and images whenever helpful—visual cues can increase accessibility for people who have trouble reading text on the screen.

5. Never signal information with sound only—someone accessing the information might not have speakers or might have hearing impairments.[1]
6. Test the application without using either a mouse or a keyboard. Access to an application's functionality should not be limited to one input device.

For more information on these and other design guidelines for accessible applications, please refer to the Visual Studio .NET documentation under the **overview** subsection of the index topic **accessibility**. This section provides links to discussions of how to design more accessible Windows and ASP.NET applications.

One specific way that programmers can make their applications more accessible is to use a *text-to-speech* control in their programs. A text-to-speech control can convert text into speech—a computerized voice speaks the words provided as text to the control. Text-to-speech controls facilitate access for people who cannot see the screen.

Another way to make applications more accessible is to use *tab stops*. A tab stop occurs when the user presses the *Tab* key, causing the focus to transfer to another control. The order in which the controls gain focus is called the *tab order*, which is determined by the ***`TabIndex`*** value of the controls (controls gain focus in ascending order). Each control also has a ***`TabStop`*** property—if this property is **`true`**, the control is included in the tab order; otherwise, it is not. Using the **`TabIndex`** and **`TabStop`** properties makes it simple to create more easily navigable applications. If these properties are set incorrectly, the logical ordering of the application might not be maintained. Consider an application that has **`TextBox`**es in which a user inputs a first name, a last name and an address. The logical tab order would take the user from the **`TextBox`** for the first name to the one for the last name and then to the one for the address.

A third and important way in which programmers can increase the accessibility of their applications is to use specific classes provided by .NET. Class **`Control`**, for example, has many properties designed for conveying information to users. These applications can then, in turn, find the required information stored as properties. Figure 24.12 lists some properties of class **`Control`** that are designed to provide information to users.

Property	**Purpose**
`AccessibleDescription`	Describes the control to an accessibility client application. For example, a **`CheckBox`** that says **`"New User"`** would not require more description, but a **`CheckBox`** with an image of a cat would have its **`AccessibleDescription`** property set to something like, **`"A CheckBox with an image of a cat on it"`**.
`AccessibleName`	Contains a short name or identifier for the control.

Fig. 24.12 Properties of class **`Control`** related to accessibility. (Part 1 of 2.)

---

[1] "Basic Principles of Accessible Design," *.NET Framework Developer's Guide*, Visual Studio .NET Online Help

Property	Purpose
`AccessibleRole`	Member of the `AccessibleRole` enumeration. Represents the role of this control in the application—this information might help the accessibility client application determine what actions it should take.
`IsAccessible`	Contains a `bool` value specifying whether the control is visible to accessibility client applications.

Fig. 24.12 Properties of class `Control` related to accessibility. (Part 2 of 2.)

The application in Fig. 24.13 uses a text-to-speech control, tab stops and class `Control`'s accessibility-related properties. It consists of a form with three `Label`s, three `TextBox`es and a `Button`, enabling a user to submit the information. Submitting the information simply terminates the application—the application is intended only to demonstrate the use of the text-to-speech control.

The accessibility features in this program work as follows: When the mouse is over a `Label`, the text-to-speech control prompts the user to enter the appropriate information in the `TextBox` located to the right of the `Label`. If the mouse is over a `TextBox`, the contents of the `TextBox` are spoken. Lastly, if the mouse is over `Button` **Submit**, the user is told that the button should be clicked to submit the information. The tab order is the following: The `TextBox`es where the user inputs the name, phone number and password, then the `Button`. The `Label`s and text-to-speech control are not included in the tab order, because the user cannot interact with them, and their inclusion would serve no purpose. The accessibility properties are set so that accessibility client applications will obtain appropriate information about the controls. Please note that only the relevant code generated by Visual Studio .NET is included in Fig. 24.13. To use the text-to-speech control, first add it to the **Toolbox**. This is accomplished by selecting **Customize Toolbox** from the **Tools** menu. The **Customize Toolbox** dialog pops up—check the box next to the **TextToSpeech Class** option. Click **OK** to dismiss the dialog box. The **VText** control now is in the **ToolBox** and can be dragged onto a form int he same way that any other control.

The application has three `Label`s that prompts for the user's name, phone number and password. Three corresponding `TextBox`es accept the user's input and, a `Button` allows the user to submit the form. Line 25 declares a text-to-speech control named `speaker`. We want the user to hear audio descriptions of controls when the mouse is located over those controls. Lines 112–139 define the `controls_MouseHover` event handler—we attach this method to the three `TextBox`es and the `Button` as the event handler for the `MouseHover` event.

```
1  // Fig. 24.13: TextToSpeech.cs
2  // Providing audio for people with visual impairments.
3
4  using System;
5  using System.Drawing;
```

Fig. 24.13 Application with accessibility features. (Part 1 of 4.)

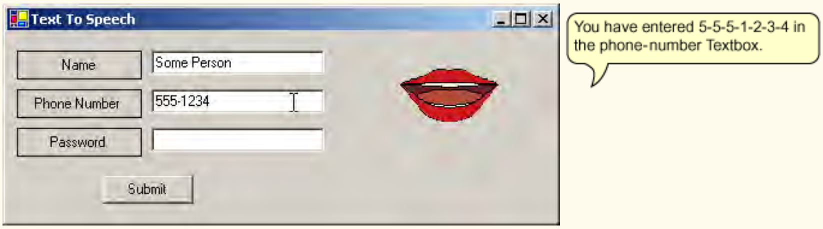
```
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// helps users navigate form with aid of audio cues
public class TextToSpeech : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label nameLabel;
   private System.Windows.Forms.Label phoneLabel;

   private System.Windows.Forms.TextBox nameTextBox;
   private System.Windows.Forms.TextBox phoneTextBox;
   private System.Windows.Forms.TextBox passwordTextBox;

   private System.Windows.Forms.Button submitButton;

   private System.Windows.Forms.Label passwordLabel;

   private AxHTTSLib.AxTextToSpeech speaker;

   private System.ComponentModel.Container components = null;

   // default constructor
   public TextToSpeech()
   {
      InitializeComponent();

      // set Form to be visible to accessibility applications
      this.IsAccessible = true;

      // let all controls be visible to accessibility applications
      foreach ( Control current in this.Controls )
         current.IsAccessible = true;
   }

   private void InitializeComponent()
   {
      this.nameLabel.AccessibleDescription = "User Name";
      this.nameLabel.AccessibleName = "User Name";
      this.nameLabel.TabIndex = 5;
      this.nameLabel.MouseHover +=
         new System.EventHandler( this.controls_MouseHover );

      this.phoneLabel.AccessibleDescription =
         "Phone Number Label";
      this.phoneLabel.AccessibleName = "Phone Number Label";
      this.phoneLabel.TabIndex = 6;
      this.phoneLabel.MouseHover +=
         new System.EventHandler( this.controls_MouseHover );

      this.nameTextBox.AccessibleDescription =
         "Enter User Name";
```

Fig. 24.13 Application with accessibility features. (Part 2 of 4.)

```
         this.nameTextBox.AccessibleName = "User Name TextBox";
         this.nameTextBox.TabIndex = 1;
         this.nameTextBox.MouseHover +=
            new System.EventHandler( this.controls_MouseHover );

         this.phoneTextBox.AccessibleDescription =
            "Enter Phone Number";
         this.phoneTextBox.AccessibleName = "Phone Number TextBox";
         this.phoneTextBox.TabIndex = 2;
         this.phoneTextBox.MouseHover +=
            new System.EventHandler( this.controls_MouseHover );

         this.passwordTextBox.AccessibleDescription =
            "Enter Password";
         this.passwordTextBox.AccessibleName = "Password TextBox";
         this.passwordTextBox.TabIndex = 3;
         this.passwordTextBox.MouseHover +=
            new System.EventHandler( this.controls_MouseHover );

         this.submitButton.AccessibleDescription =
            "Submit the Information";
         this.submitButton.AccessibleName = "Submit Information";
         this.submitButton.TabIndex = 4;
         this.submitButton.Text = "&Submit";
         this.submitButton.Click +=
            new System.EventHandler( this.submitButton_Click );
         this.submitButton.MouseHover +=
            new System.EventHandler( this.controls_MouseHover );

         this.passwordLabel.AccessibleDescription =
            "Password Label";
         this.passwordLabel.AccessibleName = "Password Label";
         this.passwordLabel.TabIndex = 7;
         this.passwordLabel.MouseHover +=
            new System.EventHandler( this.controls_MouseHover );

         this.speaker.AccessibleDescription =
            "Give Information about Form";
         this.speaker.AccessibleName = "Speaker";
         this.speaker.TabIndex = 8;
         this.speaker.TabStop = false;

         this.AccessibleDescription = "Registration Form";
         this.AccessibleName = "Registration Form";
      }

      [STAThread]
      static void Main()
      {
         Application.Run( new TextToSpeech() );
      }

```

Fig. 24.13 Application with accessibility features. (Part 3 of 4.)

```
111     // tell user over which control mouse is
112     private void controls_MouseHover(
113        object sender, System.EventArgs e )
114     {
115        // if mouse is over Label, tell user to enter information
116        if ( sender.GetType() == nameLabel.GetType() )
117        {
118           Label temporary = ( Label) sender;
119           speaker.Speak( "Please enter your " + temporary.Text +
120              " in the textbox to the right" );
121        }
122
123        // if mouse is over TextBox, tell user what
124        // information was entered
125        else if ( sender.GetType() == nameTextBox.GetType() )
126        {
127           TextBox temporary = ( TextBox ) sender;
128           speaker.Speak( "You have entered " +
129              ( temporary.Text == "" ? "nothing" :
130              temporary.Text ) + " in the " + temporary.Name );
131        }
132
133        // otherwise, user is over Button, so tell user to click
134        // it to submit information
135        else
136           speaker.Speak(
137              "Click on this button to submit your information" );
138
139     } // end method controls_MouseHover
140
141     // thank user for information submission
142     private void submitButton_Click(
143        object sender, System.EventArgs e )
144     {
145        speaker.Speak(
146           "Thank you, your information has been submitted." );
147
148        Application.Exit();
149     }
150
151  } // end class TextToSpeech
```



Some Person

555-1234

You have entered 5-5-5-1-2-3-4 in the phone-number Textbox.

Fig. 24.13 Application with accessibility features. (Part 4 of 4.)

Method **`controls_MouseHover`** determines which type of control the mouse is hovering over and generates the appropriate audio. Line 116 determines whether the type of the control calling the method is the same as that of **`nameLabel`**. Here, we use method **`GetType`** of class **`Type`**, which returns an instance of class **`Type`**; this class represents information about a particular class. We call method **`GetType`** on object **`sender`**. Event-handler argument **`sender`** is a reference to the control that triggered the event. When the condition at line 116 evaluates to **`true`** (i.e., the control that triggered the event is **`nameLabel`**), lines 118–120 execute. Line 118 casts **`sender`** to a **`Label`** (now that we know it is one) and assigns it to **`Label temporary`**. Lines 119–120 call **`speaker`**'s method **`Speak`**, which provides the **`string`** that should be converted to speech.

A similar process is performed to determine whether the mouse is over a **`TextBox`** (line 125) and to generate the appropriate audio (lines 127–130). Lastly, if the control over which the mouse is hovering is neither a **`Label`** nor a **`TextBox`**, it must be the **`Button`**; lines 136–137 tell the user to click the button to submit information. Method **`submitButton_Click`** (lines 142–149) executes when the user clicks the **`Button`**. This event handler calls **`speaker`**'s method **`Speak`**, providing as an argument a thank-you message, and then exits the application.

Line 82 sets the **`Text`** property of **`submitButton`** to **`"&Submit"`**. This is an example of providing keyboard access to the functionality of the application. Recall that, in Chapter 13, we assigned shortcut keys by placing **`"&"`** in front of the letter that would become the shortcut key. Here, we do the same for **`submitButton`**—pressing **Alt+S** on the keyboard is equivalent to clicking the **`submitButton`**.

We establish the tab order in this application by setting the **`TabIndex`** and **`TabStop`** properties. The **`TabIndex`** properties of the controls are assigned in lines 46, 60, 67, 74, 81, 91 and 98. The **`TextBox`**es are assigned the tab indices 1–3, in order of their appearance (vertically) on the form. The **`Button`** is assigned tab index 4, and the rest of the controls are given tab indices 5–8. We want the tab order to include only the **`TextBox`**es and the **`Button`**. The default setting for the **`TabStop`** property of **`Label`**s is **`false`**—thus, we do not need to change it; the labels will not be included in the tab order. The **`TabStop`** property of **`TextBox`**es and **`Button`**s is **`true`**, which means that we do not need to change the values for those controls either. The **`TabStop`** property of **`speaker`**, however, is **`true`** by default. We set it to **`false`**, indicating that we do not want **`speaker`** included in the tab order. In general, those controls with which the user cannot directly interact should have their **`TabStop`** properties set to **`false`**.

The last accessibility feature in this application involves setting the accessibility properties of the controls so that client accessibility applications can access and process the controls properly. Lines 44, 50–51, 57–58, 64–65, 71–72, 78–79, 88–89 and 95–96 set the **`AccessibleDescription`** properties of all the controls (including the Form). Lines 45, 52, 59, 66, 73, 80, 90 and 97 set the **`AccessibleName`** properties of all the controls (again including the Form). The **`IsAccessible`** property is not visible in the **Properties** window during design time, so we must write code to set it to **`true`**. Line 35 sets the **`IsAccessible`** property of **`TextToSpeech`** to **`true`**. Lines 38–39 loop through each control on the form and set each **`IsAccessible`** property to **`true`**. The Form and all its controls now will be visible to client accessibility applications.

24.8 Accessibility in XHTML Tables

Complex Web pages often contain tables that format content and present data. However, many screen readers are incapable of translating tables correctly unless developers design the tables with screen-reader requirements in mind. For example, the *CAST eReader*, a screen reader developed by the Center for Applied Special Technology (**`www.cast.org`**), starts at the top-left-hand cell and reads columns from left to right, top to bottom. This technique of reading data from a table is referred to as *linearized*. Figure 24.14 creates a simple table listing the costs of various fruits; later, we provide this table to the CAST eReader to demonstrate its linear reading of the table. The CAST eReader reads the table in Fig. 24.14 as follows:

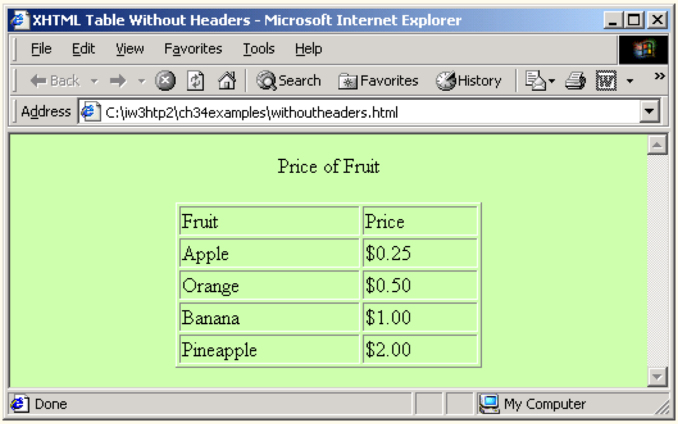
```
Price of Fruit Fruit Price Apple $0.25 Orange $0.50 Banana
$1.00 Pineapple $2.00
```

This reading does not present the content of the table adequately: The reading neither specifies caption and header information nor links data contained in cells to the column headers that describe them. WCAG 1.0 recommends using Cascading Style Sheets (CSS) instead of tables, unless a table's content linearizes in an understandable manner.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 24.14: withoutheaders.html -->
<!-- Table without headers           -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>XHTML Table Without Headers</title>

      <style type = "text/css">
         body { background-color: #ccffaa;
                text-align: center }
      </style>
   </head>

   <body>

      <p>Price of Fruit</p>

      <table border = "1" width = "50%">

         <tr>
            <td>Fruit</td>
            <td>Price</td>
         </tr>

         <tr>
            <td>Apple</td>
            <td>$0.25</td>
         </tr>

```

Fig. 24.14 XHTML table without accessibility modifications. (Part 1 of 2.)

```
            <tr>
               <td>Orange</td>
               <td>$0.50</td>
            </tr>

            <tr>
               <td>Banana</td>
               <td>$1.00</td>
            </tr>

            <tr>
               <td>Pineapple</td>
               <td>$2.00</td>
            </tr>

         </table>

      </body>
   </html>
```



XHTML Table Without Headers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\iw3htp2\ch34examples\withoutheaders.html
Done
My Computer

Price of Fruit

Fruit	Price
Apple	$0.25
Orange	$0.50
Banana	$1.00
Pineapple	$2.00

Fig. 24.14 XHTML table without accessibility modifications. (Part 2 of 2.)

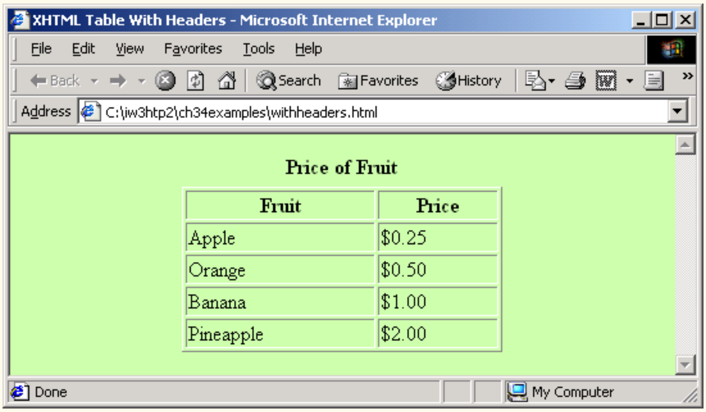
If the table in Fig. 24.14 were large, the screen reader's linearized reading would be even more confusing to users. However, modifying the **`<td>`** tag with the **`headers`** attribute and modifying *header cells* (cells specified by the **`<th>`** tag) with the **`id`** attribute causes the table to be read as intended. Figure 24.15 demonstrates how these modifications change the way in which a screen reader interprets the table.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 24.15: withheaders.html   -->
<!-- Table with headers              -->

```

Fig. 24.15 Table optimized for screen reading, using attribute **`headers`**. (Part 1 of 3.)

```
<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>XHTML Table With Headers</title>

      <style type = "text/css">
         body { background-color: #ccffaa;
                text-align: center }
      </style>
   </head>

   <body>

   <!-- This table uses the id and headers attributes to     -->
   <!-- ensure readability by text-based browsers. It also   -->
   <!-- uses a summary attribute, used by screen readers to  -->
   <!-- describe the table.                                  -->

      <table width = "50%" border = "1"
         summary = "This table uses th elements and id and
         headers attributes to make the table readable
         by screen readers">

         <caption><strong>Price of Fruit</strong></caption>

         <tr>
            <th id = "fruit">Fruit</th>
            <th id = "price">Price</th>
         </tr>

         <tr>
            <td headers = "fruit">Apple</td>
            <td headers = "price">$0.25</td>
         </tr>

         <tr>
            <td headers = "fruit">Orange</td>
            <td headers = "price">$0.50</td>
         </tr>

         <tr>
            <td headers = "fruit">Banana</td>
            <td headers = "price">$1.00</td>
         </tr>

         <tr>
            <td headers = "fruit">Pineapple</td>
            <td headers = "price">$2.00</td>
         </tr>

      </table>

   </body>
</html>
```

Fig. 24.15 Table optimized for screen reading, using attribute `headers`. (Part 2 of 3.)


XHTML Table With Headers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\iw3htp2\ch34examples\withheaders.html
Done
My Computer

Price of Fruit

Fruit	Price
Apple	$0.25
Orange	$0.50
Banana	$1.00
Pineapple	$2.00

Fig. 24.15 Table optimized for screen reading, using attribute **`headers`**. (Part 3 of 3.)

This table does not appear to be different from the standard XHTML table shown in Fig. 24.14. However, the formatting of this table allows a screen reader to read the contained data more intelligently. A screen reader vocalizes the data from the table in Fig. 24.15 as follows:

```
Caption: Price of Fruit
Summary: This table uses th elements and id and headers
attributes to make the table readable by screen readers
Fruit: Apple, Price: $0.25
Fruit: Orange, Price: $0.50
Fruit: Banana, Price: $1.00
Fruit: Pineapple, Price: $2.00
```

Every cell in the table is preceded by its corresponding header when read by the screen reader. This format helps the listener understand the table. The ***`headers`*** *attribute* is intended specifically for use in tables that hold large amounts of data. Most small tables linearize fairly well, as long as the **`<th>`** tag is used properly. We also suggest using the **`summary`** attribute and **`caption`** element to enhance clarity. To view additional examples that demonstrate how to make tables accessible, visit **`www.w3.org/TR/WCAG`**.

24.9 Accessibility in XHTML Frames

Web designers often use frames to display more than one XHTML file in a single browser window. Frames are a convenient way to ensure that certain content always displays on the screen. Unfortunately, frames often lack proper descriptions, and this prevents users with text-based browsers and users listening via speech synthesizers from navigating the Web site.

A site that uses frames must provide a meaningful description of each frame in the frame's **`<title>`** tag. Examples of good titles include "*Navigation Frame*" and "*Main Content Frame*." Users navigating via text-based browsers, such as Lynx, must choose which frame they want to open; descriptive titles make this choice simpler. However, the assignment of titles to frames does not solve all the navigation problems associated with frames. Web designers also should use the **`<noframes>`** tag, which provides alternative content for browsers that do not support frames.

Look-and-Feel Observation 24.1

Always provide titles for frames to ensure that user agents that do not support frames have alternatives.

Look-and-Feel Observation 24.2

Include a title for each frame's contents with the `frame` *element; if possible, provide links to the individual pages within the frameset, so that users still can navigate through the Web pages. To provide alternative content to browsers that do not support frames, use the* `<noframes>` *tag. This also improves access for browsers that offer limited support for frames.*

WCAG 1.0 suggests using Cascading Style Sheets (CSS) as an alternative to frames, because CSS can provide similar functionality and is highly customizible. Unfortunately, the ability to display multiple XHTML documents in a single browser window requires the complete support of HTML 4, which is not widespread. However, the second generation of Cascading Style Sheets (CSS2) can display a single document as if it were several documents. CSS2 is not yet fully supported by many user agents.

24.10 Accessibility in XML

XML gives developers the freedom to create new markup languages. Although this feature provides many advantages, the new languages might not incorporate accessibility features. To prevent the proliferation of inaccessible languages, the WAI is developing guidelines—the *XML Guidelines (XML GL)*—to facilitate the creation of accessible XML documents. The XML Guidelines recommend including a text description, similar to XHTML's `<alt>` tag, for each non-text object on a page. To enhance accessibility further, element types should allow grouping and classification and should identify important content. Without an accessible user interface, other efforts to implement accessibility are less effective. Therefore, it is essential to create stylesheets that can produce multiple outputs, including document outlines.

Many XML languages, including Synchronized Multimedia Integration Language (SMIL) and Scalable Vector Graphics (SVG), have implemented several of the WAI guidelines. The WAI XML Accessibility Guidelines can be found at `www.w3.org/WAI/PF/xmlgl.htm`.

24.11 Using Voice Synthesis and Recognition with VoiceXML™

A joint effort by AT&T®, IBM®, Lucent™ and Motorola® has created an XML vocabulary that marks up information for use by *speech synthesizers*, or tools that enable computers to speak to users. This technology, called *VoiceXML*, can provide tremendous benefits to people with visual impairments and to people who are illiterate. VoiceXML-enabled applications read Web pages to the user and then employ *speech recognition* technology to understand words spoken into a microphone. An example of a speech-recognition tool is IBM's *ViaVoice* (`www-4.ibm.com/software/speech`). To learn more about speech recognition and synthesis, consult Chapter 16, Graphics and Multimedia.

The VoiceXML interpreter and the VoiceXML browser process VoiceXML. In the future, Web browsers might incorporate these interpreters. VoiceXML is derived from XML, so VoiceXML is platform–independent. When a VoiceXML document is loaded, a *voice server* sends a message to the VoiceXML browser and begins a verbal conversation between the user and the computer.

The IBM *WebSphere Voice Server SDK 1.5* is a VoiceXML interpreter that can be used to test VoiceXML documents on the desktop. To download the VoiceServer SDK, visit **`www.alphaworks.ibm.com/tech/voiceserversdk`**. [*Note*: To run the VoiceXML program in Fig. 24.16, download *Java 2 Platform Standard Edition* (Java SDK) 1.3 from **`www.java.sun.com/j2se/1.3`**. Installation instructions for both the VoiceServerSDK and the Java SDK are located on the Deitel & Associates, Inc., Web site at **`www.deitel.com`**.]

Figure 24.16 and Fig. 24.17 depict examples of VoiceXML that could be included on a Web site. The computer speaks a document's text to the user, and the text embedded in the VoiceXML tags enables verbal interaction between the user and the browser. The output included in Fig. 24.17 demonstrates a conversation that might take place between a user and a computer after this document is loaded.

```
<?xml version = "1.0"?>
<vxml version = "1.0">

<!-- Fig. 24.16: main.vxml -->
<!-- Voice page           -->

<link next = "#home">
   <grammar>home</grammar>
</link>

<link next = "#end">
   <grammar>exit</grammar>
</link>

<var name = "currentOption" expr = "'home'"/>

<form>
   <block>
      <emp>Welcome</emp> to the voice page of Deitel and
      Associates. To exit any time say exit.
      To go to the home page any time say home.
   </block>

   <subdialog src = "#home"/>
</form>

<menu id = "home">
   <prompt count = "1" timeout = "10s">
      You have just entered the Deitel home page.
      Please make a selection by speaking one of the
      following options:
      <break msecs = "1000" />
      <enumerate/>
   </prompt>

   <prompt count = "2">
      Please say one of the following.
      <break msecs = "1000" />
```

Fig. 24.16 Home page written in VoiceXML. (Part 1 of 3.)

```
      <enumerate/>
   </prompt>

   <choice next = "#about">About us</choice>
   <choice next = "#directions">Driving directions</choice>
   <choice next = "publications.vxml">Publications</choice>
</menu>

<form id = "about">
   <block>
      About Deitel and Associates, Inc.
      Deitel and Associates, Inc. is an internationally
      recognized corporate training and publishing
      organization, specializing in programming languages,
      Internet and World Wide Web technology and object
      technology education. Deitel and Associates, Inc. is a
      member of the World Wide Web Consortium. The company
      provides courses on Java, C++, Visual Basic, C, Internet
      and World Wide Web programming and Object Technology.
      <assign name = "currentOption" expr = "'about'"/>
      <goto next = "#repeat"/>
   </block>
</form>

<form id = "directions">
   <block>
      Directions to Deitel and Associates, Inc.
      We are located on Route 20 in Sudbury,
      Massachusetts, equidistant from route
      <sayas class = "digits">128</sayas> and route
      <sayas class = "digits">495</sayas>.
      <assign name = "currentOption" expr = "'directions'"/>
      <goto next = "#repeat"/>
   </block>
</form>

<form id = "repeat">
   <field name = "confirm" type = "boolean">
      <prompt>
         To repeat say yes. To go back to home, say no.
      </prompt>

      <filled>
         <if cond = "confirm == true">
            <goto expr = "'#' + currentOption"/>
         <else/>
           <goto next = "#home"/>
         </if>
      </filled>

   </field>
</form>

```

Fig. 24.16 Home page written in VoiceXML. (Part 2 of 3.)

```
<form id = "end">
   <block>
      Thank you for visiting Deitel and Associates voice page.
      Have a nice day.
      <exit/>
   </block>
</form>

</vxml>
```

Fig. 24.16 Home page written in VoiceXML. (Part 3 of 3.)

```
<?xml version = "1.0"?>
<vxml version = "1.0">

<!-- Fig. 24.17: publications.vxml       -->
<!-- Voice page for various publications -->

<link next = "main.vxml#home">
   <grammar>home</grammar>
</link>

<link next = "main.vxml#end">
   <grammar>exit</grammar>
</link>

<link next = "#publication">
   <grammar>menu</grammar>
</link>

<var name = "currentOption" expr = "'home'"/>

<menu id = "publication">

   <prompt count = "1" timeout = "12s">
      Following are some of our publications. For more
      information visit our web page at www.deitel.com.
      To repeat the following menu, say menu at any time.
      Please select by saying one of the following books:
      <break msecs = "1000" />
      <enumerate/>
   </prompt>

   <prompt count = "2">
      Please select from the following books.
      <break msecs = "1000" />
      <enumerate/>
   </prompt>

   <choice next = "#java">Java.</choice>
   <choice next = "#c">C.</choice>
```

Fig. 24.17 Publication page of Deitel and Associates' VoiceXML page. (Part 1 of 4.)

```
   <choice next = "#cplus">C plus plus.</choice>
</menu>

<form id = "java">
   <block>
      Java How to program, third edition.
      The complete, authoritative introduction to Java.
      Java is revolutionizing software development with
      multimedia-intensive, platform-independent,
      object-oriented code for conventional, Internet,
      Intranet and Extranet-based applets and applications.
      This Third Edition of the world's most widely used
      university-level Java textbook carefully explains
      Java's extraordinary capabilities.
      <assign name = "currentOption" expr = "'java'"/>
      <goto next = "#repeat"/>
   </block>
</form>

<form id = "c">
   <block>
      C How to Program, third edition.
      This is the long-awaited, thorough revision to the
      world's best-selling introductory C book! The book's
      powerful "teach by example" approach is based on
      more than 10,000 lines of live code, thoroughly
      explained and illustrated with screen captures showing
      detailed output.World-renowned corporate trainers and
      best-selling authors Harvey and Paul Deitel offer the
      most comprehensive, practical introduction to C ever
      published with hundreds of hands-on exercises, more
      than 250 complete programs written and documented for
      easy learning, and exceptional insight into good
      programming practices, maximizing performance, avoiding
      errors, debugging, and testing. New features include
      thorough introductions to C++, Java, and object-oriented
      programming that build directly on the C skills taught
      in this book; coverage of graphical user interface
      development and C library functions; and many new,
      substantial hands-on projects.For anyone who wants to
      learn C, improve their existing C skills, and understand
      how C serves as the foundation for C++, Java, and
      object-oriented development.
      <assign name = "currentOption" expr = "'c'"/>
      <goto next = "#repeat"/>
   </block>
</form>

<form id = "cplus">
   <block>
      The C++ how to program, second edition.
      With nearly 250,000 sold, Harvey and Paul Deitel's C++
      How to Program is the world's best-selling introduction
```

Fig. 24.17 Publication page of Deitel and Associates' VoiceXML page. (Part 2 of 4.)

```
      to C++ programming. Now, this classic has been thoroughly
      updated! The new, full-color Third Edition has been
      completely revised to reflect the ANSI C++ standard, add
      powerful new coverage of object analysis and design with
      UML, and give beginning C++ developers even better live
      code examples and real-world projects. The Deitels' C++
      How to Program is the most comprehensive, practical
      introduction to C++ ever published with hundreds of
      hands-on exercises, roughly 250 complete programs written
      and documented for easy learning, and exceptional insight
      into good programming practices, maximizing performance,
      avoiding errors, debugging, and testing. This new Third
      Edition covers every key concept and technique ANSI C++
      developers need to master: control structures, functions,
      arrays, pointers and strings, classes and data
      abstraction, operator overloading, inheritance, virtual
      functions, polymorphism, I/O, templates, exception
      handling, file processing, data structures, and more. It
      also includes a detailed introduction to Standard
      Template Library containers, container adapters,
      algorithms, and iterators.
      <assign name = "currentOption" expr = "'cplus'"/>
      <goto next = "#repeat"/>
   </block>
</form>

<form id = "repeat">
   <field name = "confirm" type = "boolean">

      <prompt>
         To repeat say yes. Say no, to go back to home.
      </prompt>

      <filled>
         <if cond = "confirm == true">
            <goto expr = "'#' + currentOption"/>
         <else/>
            <goto next = "#publication"/>
         </if>
      </filled>
   </field>
</form>
</vxml>
```

Computer speaks:

```
Welcome to the voice page of Deitel and Associates. To exit any time
say exit. To go to the home page any time say home.
```

User speaks:

```
Home
```

(continued on next page)

Fig. 24.17 Publication page of Deitel and Associates' VoiceXML page. (Part 3 of 4.)

(continued from previous page)

Computer speaks:

```
You have just entered the Deitel home page. Please make a selection by
speaking one of the following options: About us, Driving directions,
Publications.
```

User speaks:

```
Driving directions
```

Computer speaks:

```
Directions to Deitel and Associates, Inc.
We are located on Route 20 in Sudbury,
Massachusetts, equidistant from route 128
and route 495.
To repeat say yes. To go back to home, say no.
```

Fig. 24.17 Publication page of Deitel and Associates' VoiceXML page. (Part 4 of 4.)

A VoiceXML document contains a series of dialogs and subdialogs, resulting in spoken interaction between the user and the computer. The `<form>` and <`menu>` tags implement the dialogs. A ***`form`*** element both presents information to the user and gathers data from the user. A ***`menu`*** element provides the user with list options and then transfers control to another dialog in response to the user's selection.

Lines 7–9 (of Fig. 24.16) use element ***`link`*** to create an active link to the home page. Attribute ***`next`*** specifies the URL to which the browser is directed when a user selects the link. Element ***`grammar`*** marks up the text that the user must speak to select the link. In the `link` element, we navigate to the element containing `id` `home` when a user speaks the word `home`. Lines 11–13 use element `link` to create a link to `id` `end` when a user speaks the word `exit`.

Lines 17–25 create a form dialog using element `form`, which collects information from the user. Lines 18–22 present introductory text. Element `block`, which can exist only within a `form` element, groups together elements that perform an action or an event. Element `emp` indicates that a section of text should be spoken with emphasis. If the level of emphasis is not specified, then the default level—*moderate*—is used. Our example uses the default level. [*Note*: To specify an emphasis level, use the `level` attribute. This attribute accepts the following values: *strong*, *moderate*, *none* and *reduced*.]

The `menu` element in line 27 enables users to select the page to which they would like to link. The ***`choice`*** element, which always is part of either a `menu` or a `form`, presents the options. The `next` attribute indicates the page that is loaded when a user makes a selection. The user selects a `choice` element by speaking the text marked up between the tags into a microphone. In this example, the first and second `choice` elements in lines 42–43 transfer control to a *local dialog* (i.e., a location within the same document) when they are selected. The third `choice` element transfers the user to the document `publications.vxml`. Lines 28–34 use element ***`prompt`*** to instruct the user to make a selection. Attribute ***`count`*** maintains a record of the number of times that a prompt is spoken (i.e., each time the computer reads a prompt, `count` increments by one). The `count` attribute transfers control to another prompt once a certain limit has been reached. Attribute `tim-`

`eout` specifies how long the program should wait after outputting the prompt for users to respond. In the event that the user does not respond before the timeout period expires, lines 36–40 provide a second, shorter prompt that reminds the user to make a selection.

When the user chooses the **`publications`** option, **`publications.vxml`** (Fig. 24.17) loads into the browser. Lines 107–113 define **`link`** elements that provide links to **`main.vxml`**. Lines 115–117 provide links to the **`menu`** element (lines 121–141), which asks users to select one of the following publications: Java, C or C++. The **`form`** elements in lines 143–217 describe books that correspond to these topics. Once the browser speaks the description, control transfers to the **`form`** element with an **`id`** attribute whose value equals **`repeat`** (lines 219–234).

Figure 24.18 provides a brief description of each VoiceXML tag that we used in the previous example (Fig. 24.17).

VoiceXML Tag	Description
`<assign>`	Assigns a value to a variable.
`<block>`	Presents information to users without any interaction between the user and the computer (i.e., the computer does not expect any input from the user).
`<break>`	Instructs the computer to pause its speech output for a specified period of time.
`<choice>`	Specifies an option in a `menu` element.
`<enumerate>`	Lists all the available options to the user.
`<exit>`	Exits the program.
`<filled>`	Contains elements that execute when the computer receives input for a `form` element from the user.
`<form>`	Gathers information from the user for a set of variables.
`<goto>`	Transfers control from one dialog to another.
`<grammar>`	Specifies grammar for the expected input from the user.
`<if>`, `<else>`, `<elseif>`	Indicates a control statement used for making logic decisions.
`<link>`	Performs a transfer of control similar to the `goto` statement, but a `link` can be executed at any time during the program's execution.
`<menu>`	Provides user options and then transfers control to other dialogs on the basis of the selected option.
`<prompt>`	Specifies text to be read to users when they must make a selection.
`<subdialog>`	Calls another dialog. After executing the subdialog, the calling dialog resumes control.
`<var>`	Declares a variable.
`<vxml>`	Top-level tag that specifies that the document should be processed by a VoiceXML interpreter.

Fig. 24.18 VoiceXML tags.

24.12 CallXML™

Another advancement benefiting people with visual impairments is *CallXML*, a voice technology created and supported by *Voxeo* (`www.voxeo.com`). CallXML creates phone-to-Web applications that control incoming and outgoing telephone calls. Examples of CallXML applications include voice mail, interactive voice-response systems and Internet call waiting. VoiceXML allows computers to read Web pages to users with visual impairments; CallXML reads Web content to users via a telephone. CallXML has important implications for individuals who do not have a computer, but do have a telephone.

When users access CallXML applications, a *text-to-speech (TTS)* engine converts text to an automated voice. The TTS engine then reads information contained within CallXML elements to the users. CallXML applications are tailored to respond to input from callers. [*Note*: Users must have a touch-tone phone to access CallXML applications.]

Typically, CallXML applications play prerecorded audio clips or text as output, requesting responses as input. An audio clip might contain a greeting that introduces callers to the application, or it might recite a menu of options, requesting that callers make a touch-tone entry. Certain applications, such as voice mail, might require both verbal and touch-tone input. Once the application receives the necessary input, it responds by invoking CallXML elements (such as `text`) that contain the information a TTS engine reads to users. If the application does not receive input within a designated time frame, it prompts the user to enter valid input.

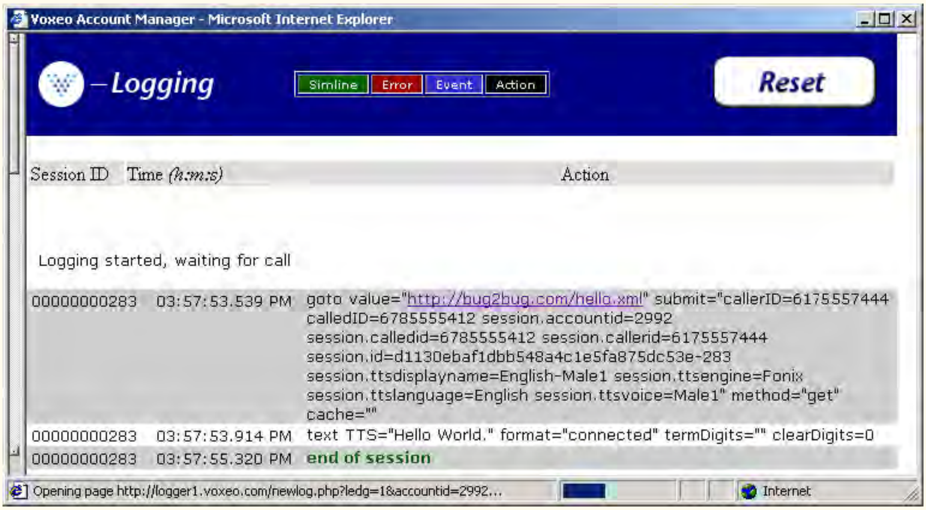
When a user accesses a CallXML application, the incoming telephone call is referred to as a *session*. A CallXML application can support multiple sessions, which means that the application can process multiple telephone calls at once. Each session is independent of the others and is assigned a unique *sessionID* for identification. A session terminates either when the user hangs up the telephone or when the CallXML application invokes the `hangup` element.

Our first CallXML application demonstrates the classic “Hello World” example (Fig. 24.19). Line 1 contains the optional *XML declaration*. Value ***`version`*** indicates the XML version to which the document conforms. The current XML recommendation is version `1.0`. Value ***`encoding`*** indicates the type of *Unicode* encoding that the application uses. For this example, we empty UTF-8, which requires eight bits to transfer and receive data. More information on Unicode can be found in Appendix G, Unicode®.

The `<callxml>` tag in line 6 declares that the content is a CallXML document. Line 7 contains the **Hello World** `text`. All text that is to be spoken by a text-to-speech (TTS) engine must be placed within `<text>` tags.

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 24.19: hello.xml          -->
4  <!-- The classic Hello World example -->
5
6  <callxml>
7     <text>Hello World.</text>
8  </callxml>
```

Fig. 24.19 **Hello World** CallXML example. (Part 1 of 2.) (Courtesy of Voxeo, © Voxeo Corporation 2000–2001.)



```
Session ID   Time (h:m:s)        Action

Logging started, waiting for call

0000000283   03:57:53.539 PM   goto value="http://bug2bug.com/hello.xml" submit="callerID=6175557444
                               calledID=6785555412 session.accountid=2992
                               session.calledid=6785555412 session.callerid=6175557444
                               session.id=d1130ebaf1dbb548a4c1e5fa875dc53e-283
                               session.ttsdisplayname=English-Male1 session.ttsengine=Fonix
                               session.ttslanguage=English session.ttsvoice=Male1" method="get"
                               cache=""
0000000283   03:57:53.914 PM   text TTS="Hello World." format="connected" termDigits="" clearDigits=0
0000000283   03:57:55.320 PM   end of session
```

Opening page http://logger1.voxeo.com/newlog.php?ledg=1&accountid=2992...

Fig. 24.19 **Hello World** CallXML example. (Part 2 of 2.) (Courtesy of Voxeo, © Voxeo Corporation 2000–2001.)

To deploy a CallXML application, register with the *Voxeo* Community (`community.voxeo.com`), a Web resource that facilitates the creation, debugging and deployment of phone applications. For the most part, Voxeo resources are free, but the company does charge fees when CallXML applications are deployed commercially. The Voxeo Community assigns a unique telephone number to each CallXML application so that external users can access and interact with the application. [*Note*: Voxeo assigns telephone numbers only to applications that reside on the Internet. If you have access to a Web server (such as IIS, PWS or Apache), use it to post your CallXML application. Otherwise, open an Internet account through one of the many Internet-service companies (such as `www.geocities.com`, `www.angelfire.com`, `www.stormpages.com`, `www.freewebsites.com`, or `www.brinkster.com`). These companies allow individuals to post documents on the Internet using their Web servers.]

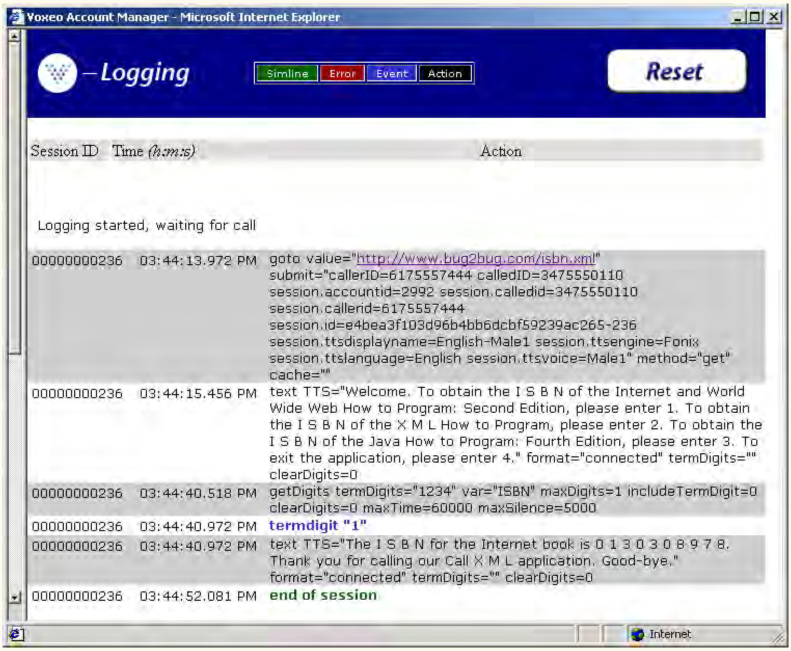
Figure 24.19 also demonstrates the *logging* feature of the **Voxeo Account Manager**, which is accessible to registered members. The logging feature records and displays the “conversation” between the user and the application. The first row of the logging feature lists the URL of the CallXML application and the *global variables* associated with that session. When a session begins, the application creates and assigns values to global variables that the entire application can access and modify. The subsequent row(s) display the “conversation.” This example demonstrates a one-way conversation (i.e., the application does not accept any input from the user) in which the TTS engine says **Hello World**. The last row displays the **end of session** message, which states that the phone call has terminated. The logging feature assists developers in the debugging of their applications. By observing a CallXML “conversation,” a developer can determine the point at which the application terminates. If the application terminates abruptly (“crashes”), the logging feature displays information regarding the type and location of the error, pointing the developer toward the section of the application that is causing the problem.

The next example (Fig. 24.20) depicts a CallXML application that reads the ISBN numbers of three Deitel textbooks—*Internet and World Wide Web How to Program: Second Edition*, *XML How to Program* and *Java How to Program: Fourth Edition*—on the basis of a user's touch-tone input. [*Note*: The code has been formatted for presentation purposes.]

```
<?xml version = "1.0" encoding = "UTF-8"?>

<!-- Fig. 24.20: isbn.xml                       -->
<!-- Reads the ISBN value of three Deitel books -->

<callxml>
   <block>
      <text>
         Welcome. To obtain the ISBN of the Internet and World
         Wide Web How to Program: Second Edition, please enter 1.
         To obtain the ISBN of the XML How to Program,
         please enter 2. To obtain the ISBN of the Java How
         to Program: Fourth Edition, please enter 3. To exit the
         application, please enter 4.
      </text>

      <!-- Obtains the numeric value entered by the user and -->
      <!-- stores it in the variable ISBN. The user has 60    -->
      <!-- seconds to enter one numeric value                 -->
      <getDigits var = "ISBN"
         maxDigits = "1"
         termDigits = "1234"
         maxTime = "60s" />

      <!-- Requests that the user enter a valid numeric -->
      <!-- value after the elapsed time of 60 seconds   -->
      <onMaxSilence>
         <text>
            Please enter either 1, 2, 3 or 4.
         </text>

         <getDigits var = "ISBN"
            termDigits = "1234"
            maxDigits = "1"
            maxTime = "60s" />

      </onMaxSilence>

      <onTermDigit value = "1">
         <text>
            The ISBN for the Internet book is 0130308978.
            Thank you for calling our CallXML application.
            Good-bye.
         </text>
      </onTermDigit>

```

Fig. 24.20 CallXML example that reads three ISBN values. (Part 1 of 2.) (Courtesy of Voxeo, © Voxeo Corporation 2000–2001.)

```
      <onTermDigit value = "2">
         <text>
            The ISBN for the XML book is 0130284173.
            Thank you for calling our CallXML application.
            Good-bye.
         </text>
      </onTermDigit>

      <onTermDigit value = "3">
         <text>
            The ISBN for the Java book is 0130341517.
            Thank you for calling our CallXML application.
            Good-bye.
         </text>
      </onTermDigit>

      <onTermDigit value = "4">
         <text>
            Thank you for calling our CallXML application.
            Good-bye.
         </text>
      </onTermDigit>
   </block>

   <!-- Event handler that terminates the call -->
   <onHangup />
</callxml>
```



```
Session ID   Time (h:m:s)                    Action

Logging started, waiting for call

00000000236   03:44:13.972 PM   goto value="http://www.bug2bug.com/isbn.xml"
                                submit="callerID=6175557444 calledID=3475550110
                                session.accountid=2992 session.calledid=3475550110
                                session.callerid=6175557444
                                session.id=e4bea3f103d96b4bb6dcbf59239ac265-236
                                session.ttsdisplayname=English-Male1 session.ttsengine=Fonix
                                session.ttslanguage=English session.ttsvoice=Male1" method="get"
                                cache=""
00000000236   03:44:15.456 PM   text TTS="Welcome. To obtain the I S B N of the Internet and World
                                Wide Web How to Program: Second Edition, please enter 1. To obtain
                                the I S B N of the X M L How to Program, please enter 2. To obtain the
                                I S B N of the Java How to Program: Fourth Edition, please enter 3. To
                                exit the application, please enter 4." format="connected" termDigits=""
                                clearDigits=0
00000000236   03:44:40.518 PM   getDigits termDigits="1234" var="ISBN" maxDigits=1 includeTermDigit=0
                                clearDigits=0 maxTime=60000 maxSilence=5000
00000000236   03:44:40.972 PM   termdigit "1"
00000000236   03:44:40.972 PM   text TTS="The I S B N for the Internet book is 0 1 3 0 3 0 8 9 7 8.
                                Thank you for calling our Call X M L application. Good-bye."
                                format="connected" termDigits="" clearDigits=0
00000000236   03:44:52.081 PM   end of session
```

Fig. 24.20 CallXML example that reads three ISBN values. (Part 2 of 2.) (Courtesy of Voxeo, © Voxeo Corporation 2000–2001.)

The ***`<block>`*** tag (line 7) encapsulates other CallXML tags. Usually, sets of CallXML tags that perform similar tasks are enclosed within **`<block>`**...**`</block>`**. The **`block`** element in this example encapsulates the ***`<text>`***, ***`<getDigits>`***, ***`<onMaxSilence>`*** and ***`<onTermDigit>`*** tags. A **`block`** element also can be nested in other **`block`** elements.

Lines 20–23 contain some attributes of the **`<getDigits>`** tag. The **`getDigits`** element obtains the user's touch-tone response and stores it in the variable declared by the ***`var`*** attribute (i.e., **ISBN**). The ***`maxDigits`*** attribute (line 21) indicates the maximum number of digits that the application can accept. This application accepts only one character. If no maximum is stated, then the application uses the default value, *nolimit*.

The ***`termDigits`*** attribute (line 22) contains the list of characters that terminate user input. When a user inputs a character from this list, the application is notified that it has received the last acceptable input; any character entered after this point is invalid. These characters do not terminate the call; they simply notify the application to proceed to the next instruction, because the necessary input has been received. In our example, the values for **`termDigits`** are **`1`**, **`2`**, **`3`** and **`4`**. The default value for **`termDigits`** is the null value (**`""`**).

The ***`maxTime`*** attribute (line 23) indicates the maximum amount of time that the application will wait for a user response. If the user fails to enter input within the given time frame, then the CallXML application invokes the event handler **`onMaxSilence`**. The default value for this attribute is 30 seconds.

The ***`onMaxSilence`*** element (lines 27–37) is an event handler that is invoked when attribute **`maxTime`** (or **`maxSilence`**) expires. The event handler notifies the application of the appropriate action to perform when a user fails to respond. In this case, the application asks the user to enter a value, because the **`maxTime`** has expired. After receiving input, **`getDigits`** (line 32) stores the entered value in the **ISBN** variable.

The ***`onTermDigit`*** element (lines 39–68) is an event handler that notifies the application of the appropriate action to perform when a user selects one of the **`termDigits`** characters. At least one **`<onTermDigit>`** tag must be associated with (i.e., must appear after) the **`getDigits`** element, even if the default value (**`""`**) is used. We provide four actions that the application can perform in response to the specific **`termDigits`** value entered by the user. For example, if the user enters **`1`**, the application reads the ISBN value for the *Internet and World Wide Web How to Program: Second Edition* textbook.

Line 72 contains the ***`<onHangup/>`*** event handler, which terminates the telephone call when the user hangs up the telephone. Our **`<onHangup>`** event handler is an empty tag (i.e., no action is performed when this tag is invoked).

The logging feature (Fig. 24.20) displays the "conversation" between the application and the user. As in the previous example, the first row specifies the URL of the application and the global variables of the session. The subsequent rows display the "conversation": The application asks the caller which ISBN value to read; the caller enters **`1`** (*Internet and World Wide Web How to Program: Second Edition*), and the application reads the corresponding ISBN. The **end of session** message states that the application has terminated.

We provide brief descriptions of various logic and action CallXML elements in Fig. 24.21. *Logic elements* assign values to, and clear values from, the session variables; *action elements* perform specified tasks, such as answering and terminating a telephone call during the current session. A complete list of CallXML elements is available at:

`www.oasis-open.org/cover/callxmlv2.html`

24.13 JAWS® for Windows

JAWS (Job Access with Sound) is one of the leading screen readers currently on the market. Henter-Joyce, a division of Freedom Scientific™, created this application to help people with visual impairments interact with technology.

To download a demonstration version of JAWS, visit **`www.freedomscientific.com`**. The JAWS demo is fully functional and includes an extensive, highly customized help system. Users can select the voice that "reads" Web content and the rate at which text is spoken. Users also can create keyboard shortcuts. Although the demo is in English, the full version of JAWS allows the user to choose one of several supported languages.

JAWS also includes special key commands for popular programs, such as Microsoft Internet Explorer and Microsoft Word. For example, when browsing in Internet Explorer, JAWS' capabilities extend beyond the reading of content on the screen. If JAWS is enabled, pressing *Insert* + *F7* in Internet Explorer opens a **Links List** dialog, which displays all the links available on a Web page. For more information about JAWS and the other products offered by Henter-Joyce, visit **`www.freedomscientific.com`**.

Elements	Description
`assign`	Assigns a **`value`** to a variable, **`var`**.
`clear`	Clears the contents of the **`var`** attribute.
`clearDigits`	Clears all digits that the user has entered.
`goto`	Navigates to another section of the current CallXML application or to a different CallXML application. The **`value`** attribute specifies the URL of the invoked application. The **`submit`** attribute lists the variables that are passed to the invoked application. The **`method`** attribute states whether to use the HTTP *get* or *post* request type when sending and retrieving information. A *get* request retrieves data from a Web server without modifying the contents, whereas the *post* request receives modified data.
`run`	Starts a new CallXML session for each call. The **`value`** attribute specifies the CallXML application to retrieve. The **`submit`** attribute lists the variables that are passed to the invoked application. The **`method`** attribute states whether to use the HTTP *get* or *post* request type. The **`var`** attribute stores the identification number of the session.
`sendEvent`	Allows multiple sessions to exchange messages. The **`value`** attribute stores the message, and the **`session`** attribute specifies the identification number of the session that receives the message.
`answer`	Answers an incoming telephone call.
`call`	Calls the URL specified by the **`value`** attribute. The **`callerID`** attribute contains the phone number that is displayed on a CallerID device. The **`maxTime`** attribute specifies the length of time to wait for the call to be answered before disconnecting.

Fig. 24.21 CallXML elements. (Part 1 of 2.)

Elements	Description
`conference`	Connects multiple sessions so that individuals can participate in a conference call. The **`targetSessions`** attribute specifies the identification numbers of the sessions, and the **`termDigits`** attribute indicates the touch-tone keys that terminate the call.
`wait`	Waits for user input. The **`value`** attribute specifies how long to wait. The **`termDigits`** attribute indicates the touch-tone keys that terminate the **`wait`** element.
`play`	Plays an audio file or pronounces a value that is stored as a number, date or amount of money and is indicated by the **`format`** attribute. The **`value`** attribute contains the information (location of the audio file, number, date or amount of money) that corresponds to the **`format`** attribute. The **`clearDigits`** attribute specifies whether or not to delete the previously entered input. The **`termDigits`** attribute indicates the touch-tone keys that terminate the audio file, etc.
`recordAudio`	Records an audio file and stores it at the URL specified by **`value`**. The **`format`** attribute indicates the file extension of the audio clip. Other attributes include **`termDigits`**, **`clearDigits`**, **`maxTime`** and **`maxSilence`**.

Fig. 24.21 CallXML elements. (Part 2 of 2.)

24.14 Other Accessibility Tools

Many accessibility products are available to assist people with disabilities. One such technology, Microsoft's *Active Accessibility*®, establishes a protocol by which an accessibility aid can retrieve information about an application's user interface in a consistent manner. Accessibility aids require information such as the name, location and layout of particular GUI elements within an application, so that the accessibility aid can render the information properly to the intended audience. Active Accessibility also enables software developers and accessibility-aid developers to design programs and products that are compatible with each other. Moreover, Active Accessibility is packaged in two components, enabling both programmers and individuals who use accessibility aids to employ the software. The *Software Development Kit (SDK)* component is intended for programmers: It includes testing tools, programmatic libraries and header files. The *Redistribution Kit (RDK)* component is intended for those who use accessibility aids: It installs a runtime component into the Microsoft operating system. Accessibility aids use the Active Accessibility runtime component to interact with and obtain information from any application software. For more information on Active Accessibility, visit:

`www.microsoft.com/enable/msaa`

Another important accessibility tool for individuals with visual impairments is the *braille keyboard*. In addition to providing keys labeled with the letters they represent, a braille keyboard also has the equivalent braille symbol printed on each key. Most often, braille keyboards are combined with a speech synthesizer or a braille display, enabling users to interact with the computer to verify that their typing is correct.

Speech synthesis also provides benefits to people with disabilities. *Speech synthesizers* have been used for many years to aid people who are unable to communicate verbally. However, the growing popularity of the Web has prompted a surge of interest in the fields of speech synthesis and speech recognition. Now, these technologies are allowing individuals with disabilities to use computers more than ever before. The development of speech synthesizers also is enabling the improvement of other technologies, such as VoiceXML and *AuralCSS* (**`www.w3.org/TR/REC-CSS2/aural.html`**). These tools allow people with visual impairments and illiterate people to access Web sites.

Despite the existence of adaptive software and hardware for people with visual impairments, the accessibility of computers and the Internet is still hampered by the high costs, rapid obsolescence and unnecessary complexity of current technology. Moreover, almost all software currently available requires installation by a person who can see. *Ocularis* is a project launched in the open-source community that aims to address these problems. (Open-source software for people with visual impairments already exists; although it is often superior to its proprietary, closed-source counterparts, it has not yet reached its full potential.) Ocularis ensures that the blind can access and use all aspects of the Linux operating system. Products that integrate with Ocularis include word processors, calculators, basic finance applications, Internet browsers and e-mail clients. In addition, a screen reader is included for use with programs that have a command-line interface. The official Ocularis Web site is located at

`ocularis.sourceforge.net`.

People with visual impairments are not the only beneficiaries of efforts to improve markup languages. People with hearing impairments also have a number of tools to help them interpret auditory information delivered over the Web. One of these tools, *Synchronized Multimedia Integration Language* (SMIL™), is designed to add extra *tracks (*layers of content found within a single audio or video file) to multimedia content. The additional tracks can contain closed captioning.

Technologies are being designed to help people with severe disabilities, such as quadriplegia, a form of paralysis that affects the body from the neck down. One such technology, *EagleEyes*, developed by researchers at Boston College (**`www.bc.edu/eagleeyes`**), is a system that translates eye movements into mouse movements. A user moves the mouse cursor by moving his or her eyes or head and is thereby able to control the computer.

GW Micro, Henter-Joyce and Adobe Systems, Inc., also are working on software that assists people with disabilities. Adobe Acrobat 5.0 complies with Microsoft's application programming interface (API) to allow businesses to provide information to a wider audience. JetForm Corp is also accommodating the needs of people with disabilities by developing server-based XML software. The new software allows users to download information in a format that best meets their needs.

There are many services on the Web that assist e-businesses in designing Web sites so that they are accessible to individuals with disabilities. For additional information, the U.S. Department of Justice (**`www.usdoj.gov`**) provides extensive resources detailing legal and technical issues related to people with disabilities.

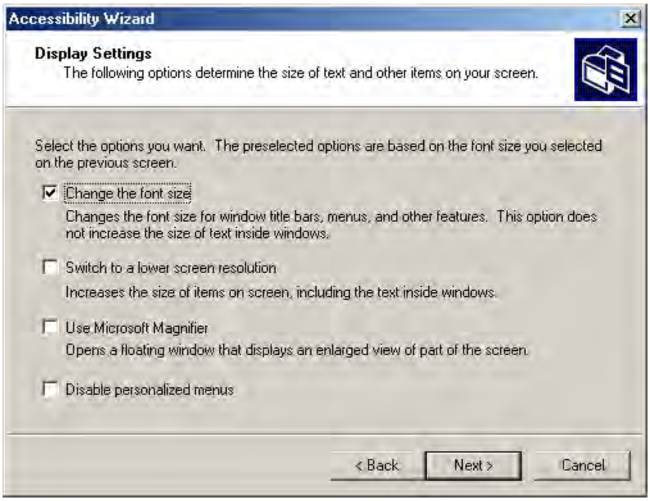
24.15 Accessibility in Microsoft® Windows® 2000

Because of the prominence of the Windows operating system, it is crucial that this operating system provide proper accessibility to individuals with disabilities. Beginning with Microsoft *Windows 95*, Microsoft has included accessibility features in its operating systems and many of its applications, including *Office 97*, *Office 2000* and *Netmeeting*. In Microsoft *Windows 2000*, Microsoft significantly enhanced the operating system's accessibility features. All the accessibility options provided by Windows 2000 are available through the ***Accessibility Wizard***, which guides users through Windows 2000 accessibility features and then configures users' computers in accordance with the chosen specifications. This section uses the **Accessibility Wizard** to guide users through the configuration of their Windows 2000 accessibility options.

To access the **Accessibility Wizard**, users' computers must be equipped with Microsoft Windows 2000. Click the **Start** button and select **Programs**, followed by **Accessories**, **Accessibility** and **Accessibility Wizard**. When the wizard starts, the **Welcome** screen displays. Click **Next**. The next dialog (Fig. 24.22) asks the user to select a font size. Modify the font size if necessary and then click **Next**.

Figure 24.22 depicts the **Display Settings** dialog. This dialog allows the user to activate the font-size settings chosen in the previous window, change the screen resolution, enable the *Microsoft Magnifier* (a program that displays an enlarged section of the screen in a separate window) and disable personalized menus. Personalized menus hide rarely used programs from the start menu and can be a hindrance to users with disabilities. Make appropriate selections and click **Next**.

The **Set Wizard Options** dialog (Fig. 24.23) asks questions about the user's disabilities; the answers to these questions allow the **Accessibility Wizard** to customize Windows to better suit the user's needs. For demonstration purposes, we selected every type of disability included in the dialogue. Click **Next** to continue.



Display Settings

The following options determine the size of text and other items on your screen.

Select the options you want. The preselected options are based on the font size you selected on the previous screen.

☑ Change the font size

Changes the font size for window title bars, menus, and other features. This option does not increase the size of text inside windows.

☐ Switch to a lower screen resolution

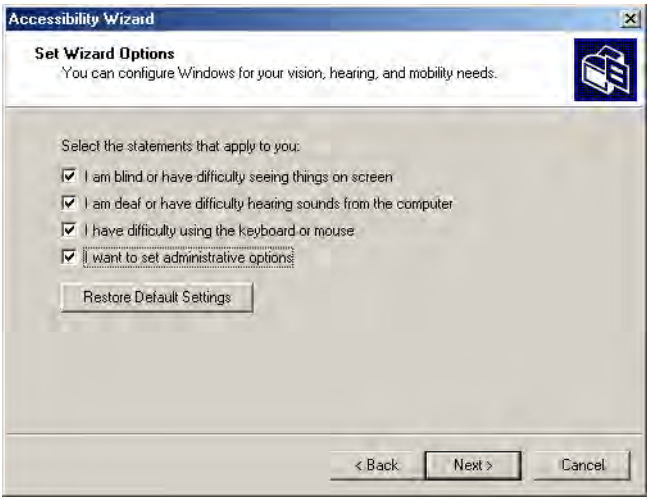
Increases the size of items on screen, including the text inside windows.

☐ Use Microsoft Magnifier

Opens a floating window that displays an enlarged view of part of the screen.

☐ Disable personalized menus

Fig. 24.22 **Display Settings** dialog.



Accessibility Wizard

Set Wizard Options

You can configure Windows for your vision, hearing, and mobility needs.

Select the statements that apply to you:

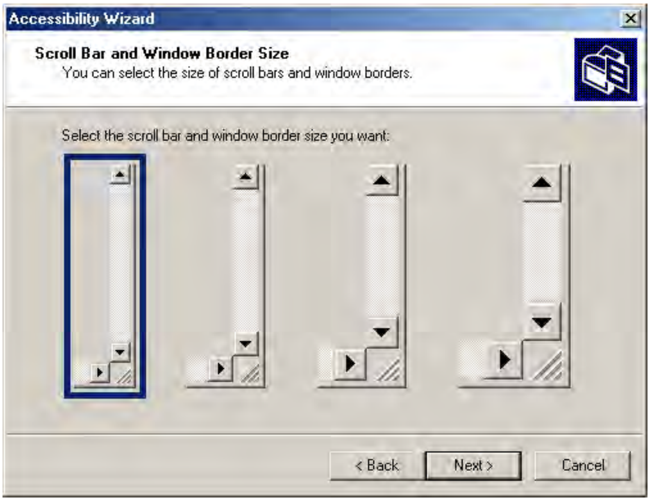
- ☑ I am blind or have difficulty seeing things on screen
- ☑ I am deaf or have difficulty hearing sounds from the computer
- ☑ I have difficulty using the keyboard or mouse
- ☑ I want to set administrative options

Fig. 24.23 **Accessibility Wizard** initialization options.

24.15.1 Tools for People with Visual Impairments

When we check all the options in Fig. 24.23, the wizard begins to configure Windows so that it is accessible to people with visual impairments. The dialog box shown in Fig. 24.24 allows the user to resize the scroll bars and window borders to increase their visibility. Click **Next** to proceed to the next dialog.

Figure 24.25 contains a dialog that allows the user to resize icons. Users with poor vision and users who are illiterate or have trouble reading benefit from large icons.



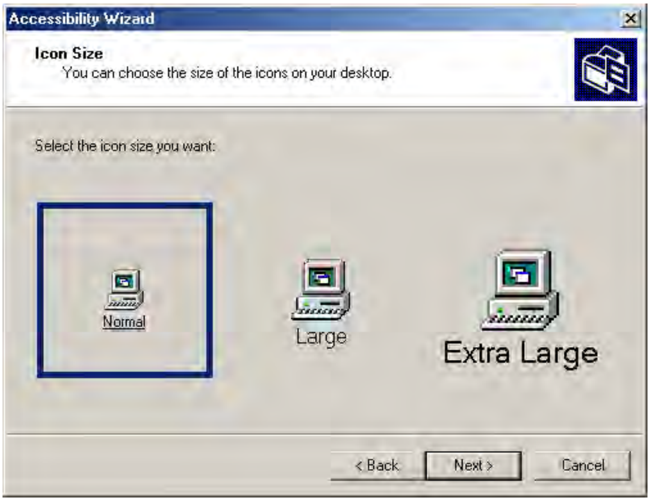
Accessibility Wizard

Scroll Bar and Window Border Size

You can select the size of scroll bars and window borders.

Select the scroll bar and window border size you want:

Fig. 24.24 Scroll Bar and Window Border Size dialog.



Accessibility Wizard

Icon Size

You can choose the size of the icons on your desktop.

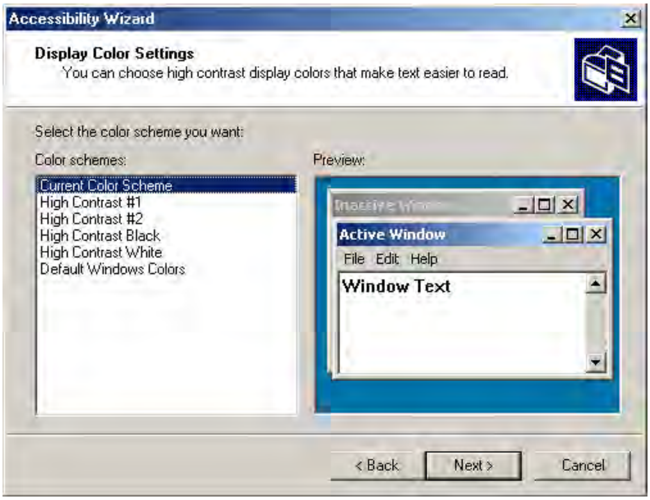
Select the icon size you want:

Normal
Large
Extra Large
< Back
Next >
Cancel

Fig. 24.25 Adjusting window-element sizes.

Clicking **Next** displays the **Display Color Settings** dialog (Fig. 24.26). These settings enable the user to change the Windows color scheme and resize various screen elements.

Click **Next** to view the dialog (Fig. 24.27) that enables customization of the mouse cursor. Anyone who has ever used a laptop computer knows how difficult it can be to see the mouse cursor. This is even more problematic for people with visual impairments. To address this problem, the wizard offers users the options of larger cursors, black cursors and cursors that invert the colors of objects underneath them. Click **Next**.



Accessibility Wizard

Display Color Settings

You can choose high contrast display colors that make text easier to read.

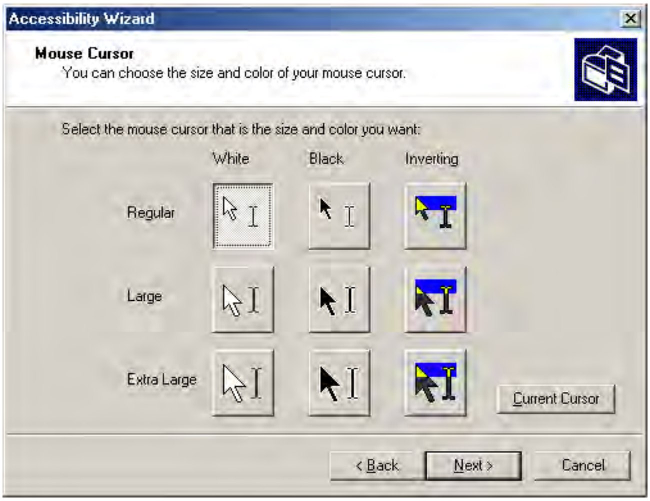
Select the color scheme you want:

Color schemes:

Current Color Scheme
High Contrast #1
High Contrast #2
High Contrast Black
High Contrast White
Default Windows Colors

Preview:
Inactive Window
Active Window
File Edit Help
Window Text
< Back
Next >
Cancel

Fig. 24.26 **Display Color Settings** options.



Accessibility Wizard
White
Black
Inverting
Regular
Large
Extra Large
Current Cursor
< Back
Next >
Cancel

Mouse Cursor

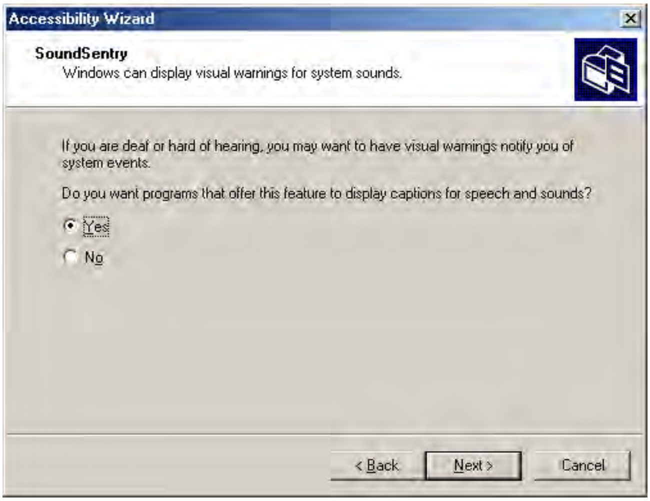
You can choose the size and color of your mouse cursor.

Select the mouse cursor that is the size and color you want:

Fig. 24.27 **Accessibility Wizard** mouse cursor adjustment tool.

24.15.2 Tools for People with Hearing Impairments

This section, which focuses on accessibility for people with hearing impairments, begins with the ***SoundSentry*** window (Fig. 24.28). **SoundSentry** is a tool that creates visual signals to notify users of system events. For example, people with hearing impairments are unable to hear the beeps that normally indicate warnings, so **SoundSentry** flashes the screen when a beep occurs. To continue on to the next dialog, click **Next**.



Accessibility Wizard
< Back
Next >
Cancel

SoundSentry

Windows can display visual warnings for system sounds.

If you are deaf or hard of hearing, you may want to have visual warnings notify you of system events.

Do you want programs that offer this feature to display captions for speech and sounds?

● Yes

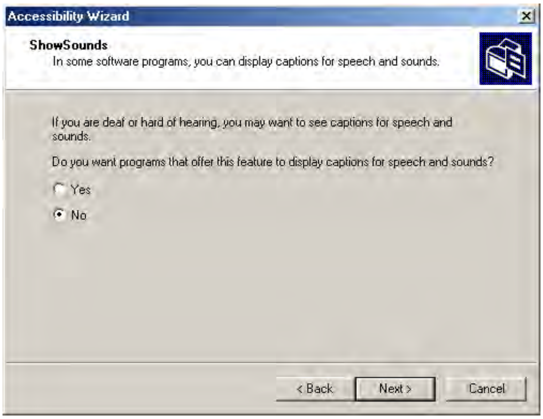
○ No

Fig. 24.28 **SoundSentry** dialog.

The next window is the ***ShowSounds*** window (Fig. 24.29). **ShowSounds** adds captions to spoken text and other sounds produced by today's multimedia-rich software. Note that, for **ShowSounds** to work in a specific application, developers must provide the captions and spoken text specifically within their software. Make selections and click **Next**.

24.15.3 Tools for Users Who Have Difficulty Using the Keyboard

The next dialog describes **StickyKeys** (Fig. 24.30). ***StickyKeys*** is a program that helps users who have difficulty pressing multiple keys at the same time. Many important computer commands can be invoked only by pressing specific key combinations. For example, the reboot command requires the user to press *Ctrl+Alt+Delete* simultaneously. **StickyKeys** enables the user to press key combinations in sequence, rather than at the same time. Click **Next** to continue to the **BounceKeys** dialog (Fig. 24.31).



Accessibility Wizard

ShowSounds

In some software programs, you can display captions for speech and sounds.

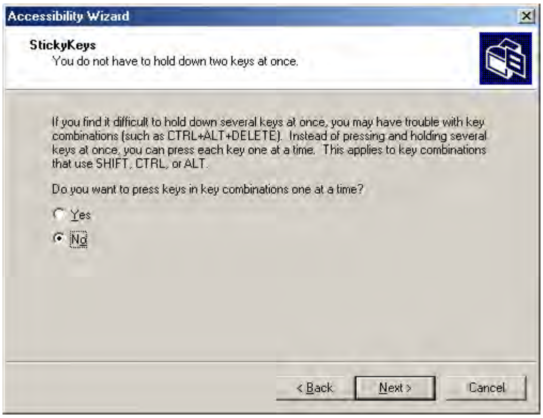
If you are deaf or hard of hearing, you may want to see captions for speech and sounds.

Do you want programs that offer this feature to display captions for speech and sounds?

○ Yes

● No

Fig. 24.29 **ShowSounds** dialog.



Accessibility Wizard

StickyKeys

You do not have to hold down two keys at once.

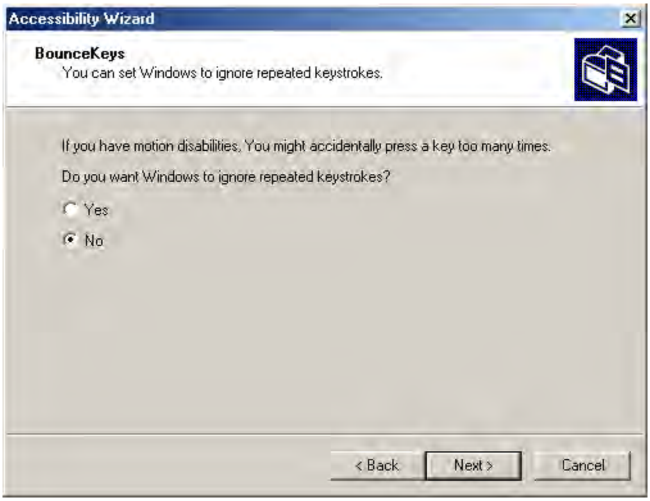
If you find it difficult to hold down several keys at once, you may have trouble with key combinations (such as CTRL+ALT+DELETE). Instead of pressing and holding several keys at once, you can press each key one at a time. This applies to key combinations that use SHIFT, CTRL, or ALT.

Do you want to press keys in key combinations one at a time?

○ Yes

● No

Fig. 24.30 **StickyKeys** window.



You can set Windows to ignore repeated keystrokes.

If you have motion disabilities, You might accidentally press a key too many times.

Do you want Windows to ignore repeated keystrokes?

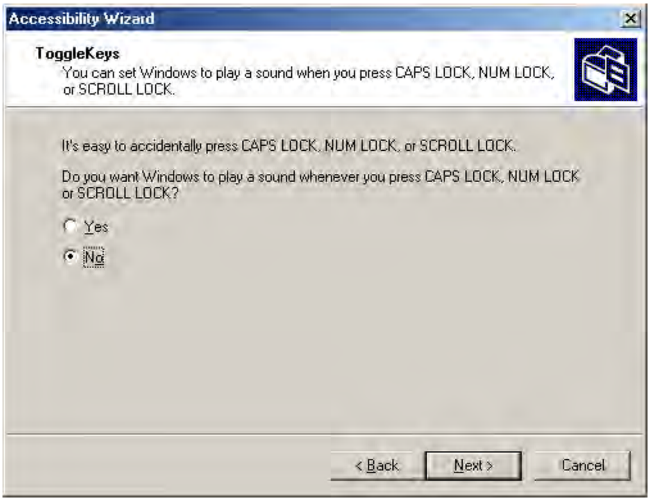
○ Yes

● No

Fig. 24.31 **BounceKeys** dialog.

Another common problem that affects certain users with disabilities is the accidental pressing of the same key multiple times. This problem typically is caused by holding a key down too long. ***BounceKeys*** forces the computer to ignore repeated keystrokes. Click **Next**.

ToggleKeys (Fig. 24.32) alerts users that they have pressed one of the lock keys (i.e., *Caps Lock*, *Num Lock* or *Scroll Lock*) by sounding an audible beep. Make selections and click **Next**.



You can set Windows to play a sound when you press CAPS LOCK, NUM LOCK, or SCROLL LOCK.

It's easy to accidentally press CAPS LOCK, NUM LOCK, or SCROLL LOCK.

Do you want Windows to play a sound whenever you press CAPS LOCK, NUM LOCK or SCROLL LOCK?

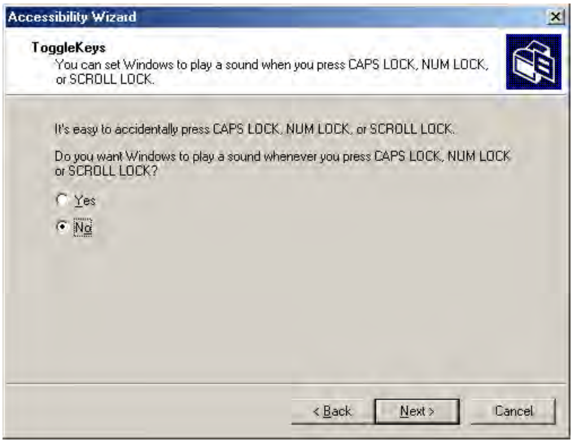
○ Yes

● No

Fig. 24.32 **ToggleKeys** window.

Next, the **Extra Keyboard Help** dialog (Fig. 24.33) is displayed. This dialog can activate a tool that displays information such as keyboard shortcuts and tool tips when such information is available. Like **ShowSounds**, this tool requires that software developers provide the content to be displayed.

Clicking **Next** will load the **MouseKeys** (Fig. 24.34) customization window. ***MouseKeys*** is a tool that uses the keyboard to imitate mouse movements. The arrow keys direct the mouse, and the *5* key indicates a single click. To double click, the user must press the + key; to simulate the holding down of the mouse button, the user must press the *Ins* (Insert) key. To release the mouse button, the user must press the *Del* (Delete) key. Choose whether to enable **MouseKeys** and then click **Next.**



ToggleKeys

You can set Windows to play a sound when you press CAPS LOCK, NUM LOCK, or SCROLL LOCK.

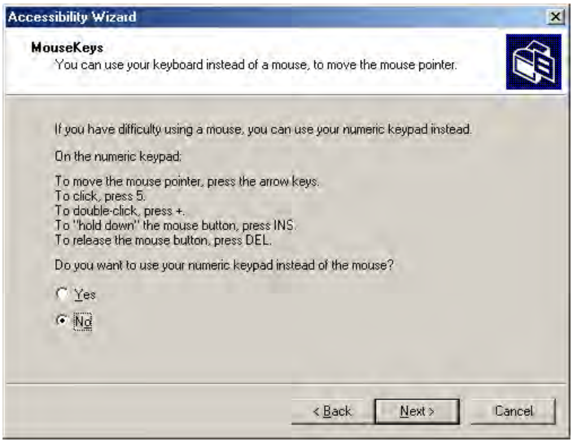
It's easy to accidentally press CAPS LOCK, NUM LOCK, or SCROLL LOCK.

Do you want Windows to play a sound whenever you press CAPS LOCK, NUM LOCK or SCROLL LOCK?

○ Yes

● No

Fig. 24.33 **Extra Keyboard Help** dialog.



MouseKeys

You can use your keyboard instead of a mouse, to move the mouse pointer.

If you have difficulty using a mouse, you can use your numeric keypad instead.

On the numeric keypad:

To move the mouse pointer, press the arrow keys.
To click, press 5.
To double-click, press +.
To "hold down" the mouse button, press INS.
To release the mouse button, press DEL.

Do you want to use your numeric keypad instead of the mouse?

○ Yes

● No

Fig. 24.34 **MouseKeys** window.

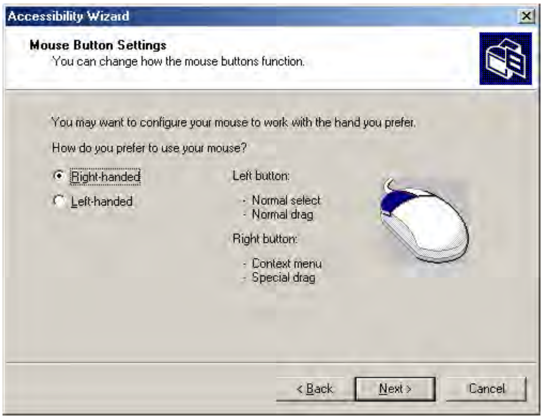
Today’s computer tools, including most mice, are designed almost exclusively for right-handed users. Microsoft recognized this problem and added the ***Mouse Button* Settings** window (Fig. 24.35) to the **Accessibility Wizard**. This tool allows the user to create a virtual left-handed mouse by swapping the button functions. Click **Next**.

Users can adjust mouse speed through the **MouseSpeed** (Fig. 24.36) section of the **Accessibility Wizard**. Dragging the scroll bar changes the speed. Clicking the **Next** button sets the speed and displays the wizard’s **Set Automatic Timeouts** window (Fig. 24.37). Although accessibility tools are important to users with disabilities, they can be a hindrance to users who do not need them. In situations where varying accessibility needs exist, it is important that the user be able to turn the accessibility tools on and off as necessary. The ***Set Automatic Timeouts*** window specifies a *timeout* period for enabling or disabling accessibility tools. A timeout either enables or disables a certain action after the computer has idled for a specified amount of time. A screen saver is a common example of a program with a timeout period. Here, a timeout is set to toggle the accessibility tools.

After the user clicks **Next**, the **Save Settings to File** dialog appears (Fig. 24.38). This dialog determines whether the accessibility settings should be used as the *default settings*, which are loaded when the computer is rebooted or after a timeout. Set the accessibility settings as the default if the majority of users needs them. Users also can save multiple accessibility settings. The user can create an `.acw` file, which, when chosen, activates the saved accessibility settings on any Windows 2000 computer.

24.15.4 Microsoft Narrator

Microsoft Narrator is a text-to-speech program designed for people with visual impairments. It reads text, describes the current desktop environment and alerts the user when certain Windows events occur. **Narrator** is intended to aid in the configuration of Microsoft Windows. It is a screen reader that works with Internet Explorer, Wordpad, Notepad and most programs in the **Control Panel**. Although its capabilities are limited outside these applications, **Narrator** is excellent at navigating the Windows environment.



Accessibility Wizard

Mouse Button Settings

You can change how the mouse buttons function.

You may want to configure your mouse to work with the hand you prefer.

How do you prefer to use your mouse?

● Right-handed

○ Left-handed

Left button:

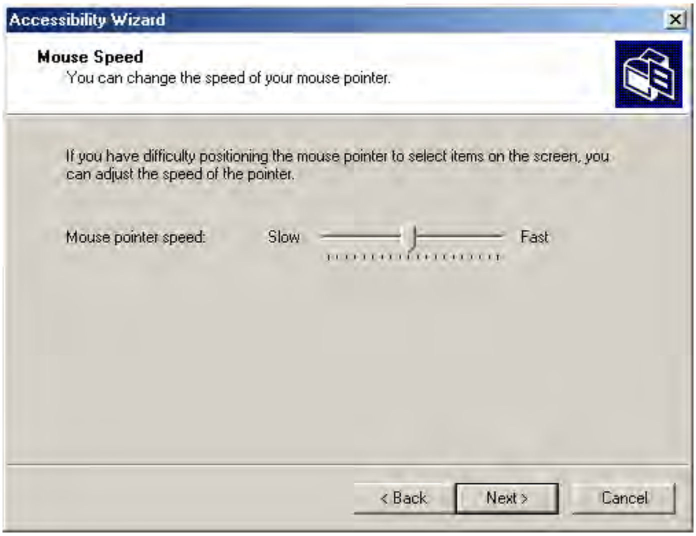
- Normal select
- Normal drag

Right button:

- Context menu
- Special drag

Fig. 24.35 **Mouse Button Settings** window.

To explore **Narrator**’s functionality, we explain how to use the program in conjunction with several Windows applications. Click the **Start** button and select **Programs**, followed by **Accessories**, **Accessibility** and **Narrator**. Once **Narrator** is open, it describes the current foreground window. It then reads the text inside the window aloud to the user. When the user clicks **OK**, the dialog in Fig. 24.39 displays.



Accessibility Wizard

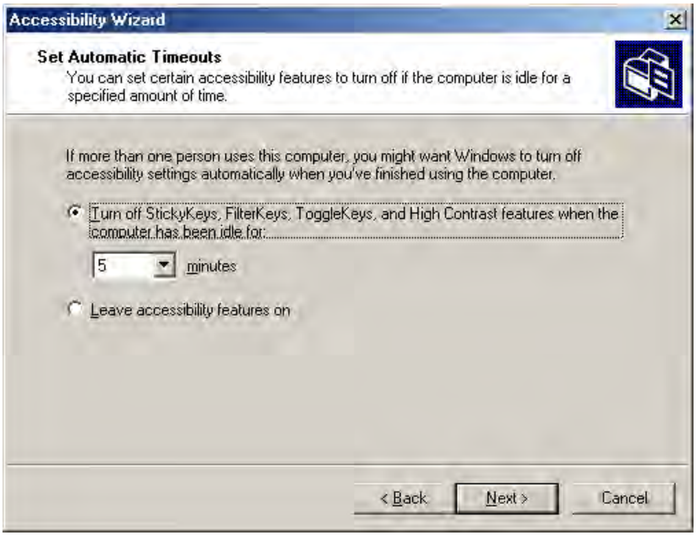
Mouse Speed

You can change the speed of your mouse pointer.

If you have difficulty positioning the mouse pointer to select items on the screen, you can adjust the speed of the pointer.

Mouse pointer speed: Slow Fast

Fig. 24.36 **Mouse Speed** dialog.



Accessibility Wizard

Set Automatic Timeouts

You can set certain accessibility features to turn off if the computer is idle for a specified amount of time.

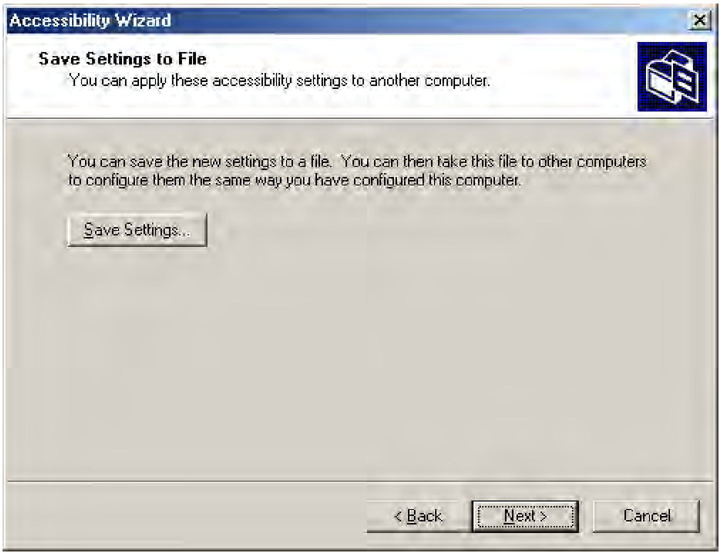
If more than one person uses this computer, you might want Windows to turn off accessibility settings automatically when you've finished using the computer.

● Turn off StickyKeys, FilterKeys, ToggleKeys, and High Contrast features when the computer has been idle for:

5 minutes

○ Leave accessibility features on

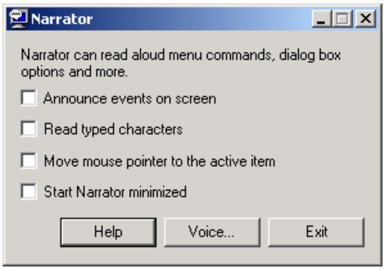
Fig. 24.37 **Set Automatic Timeouts** dialog.



Save Settings to File
You can apply these accessibility settings to another computer.

You can save the new settings to a file. You can then take this file to other computers to configure them the same way you have configured this computer.

Fig. 24.38 Saving new accessibility settings.



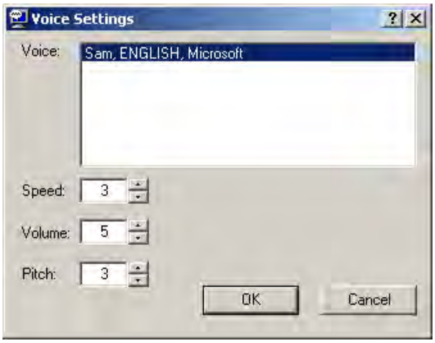
Narrator can read aloud menu commands, dialog box options and more.

☐ Announce events on screen
☐ Read typed characters
☐ Move mouse pointer to the active item
☐ Start Narrator minimized

Fig. 24.39 **Narrator** window.

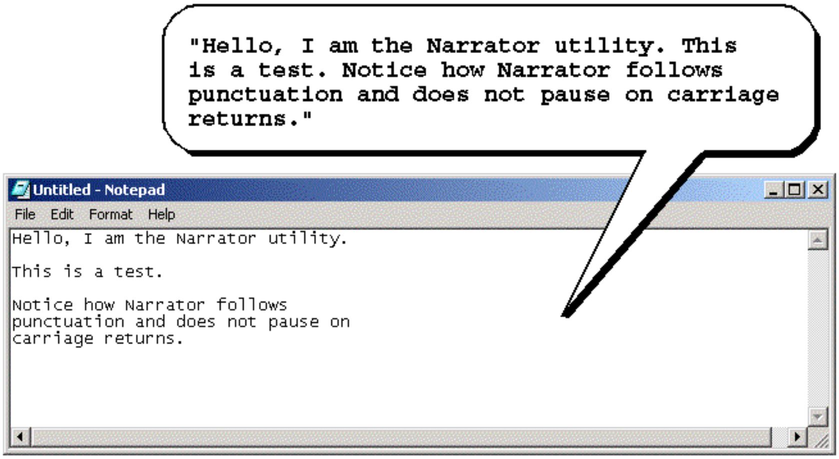
Checking the first option instructs **Narrator** to describe menus and new windows when they are opened. The second option instructs **Narrator** to speak the characters that users type as they type them. The third option moves the mouse cursor to the region currently being read by **Narrator**. Clicking the **Voice...** button enables the user to change the pitch, volume and speed of the narrator voice (Fig. 24.40).

Now, we demonstrate **Narrator** in various applications. When **Narrator** is running, open **Notepad** and click the **File** menu. **Narrator** announces the opening of the program and begins to describe the items in the **File** menu. As a user scrolls down the list, **Narrator** reads the item to which the mouse currently is pointing. Type some text and press *Ctrl-Shift-Enter* to hear **Narrator** read it (Fig. 24.41). If the **Read typed characters** option is checked, **Narrator** reads each character as it is typed. Users also can employ the keyboard's direction arrows to make **Narrator** read. The up and down arrows cause **Narrator** to speak the lines adjacent to the current mouse position, and the left and right arrows cause **Narrator** to speak the characters adjacent to the current mouse position.


Voice Settings
OK
Cancel

Voice: Sam, ENGLISH, Microsoft
Speed: 3
Volume: 5
Pitch: 3

Fig. 24.40 Voice-settings window.


"Hello, I am the Narrator utility. This is a test. Notice how Narrator follows punctuation and does not pause on carriage returns."
Untitled - Notepad
File Edit Format Help

Hello, I am the Narrator utility.

This is a test.

Notice how Narrator follows
punctuation and does not pause on
carriage returns.

Fig. 24.41 **Narrator** reading **Notepad** text.

24.15.5 Microsoft On-Screen Keyboard

Some computer users lack the ability to use a keyboard, but are able to use a pointing device, such as a mouse. For these users, the ***On-Screen Keyboard*** is helpful. To access the On-Screen Keyboard, click the **Start** button and select **Programs**, followed by **Accessories**, **Accessibility** and **On-Screen Keyboard**. Figure 24.42 depicts the layout of the Microsoft On-Screen Keyboard.

Users who have difficulty using the On-Screen Keyboard can purchase more sophisticated products, such as *Clicker 4*™ by *Inclusive Technology*. Clicker 4 is an aid designed for people who cannot use a keyboard effectively. Its best feature is that it can be customized. Keys can have letters, numbers, entire words or even pictures on them. For more information regarding Clicker 4, visit **`www.inclusive.co.uk/catalog/clicker.htm`**.

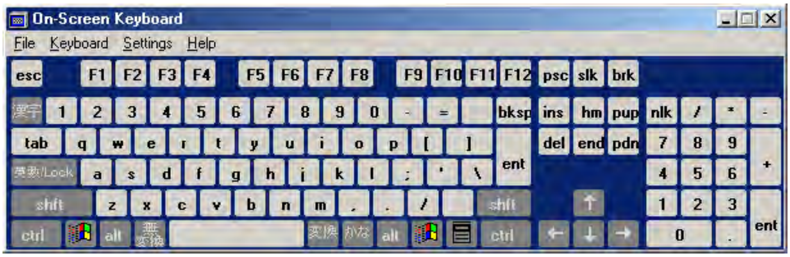
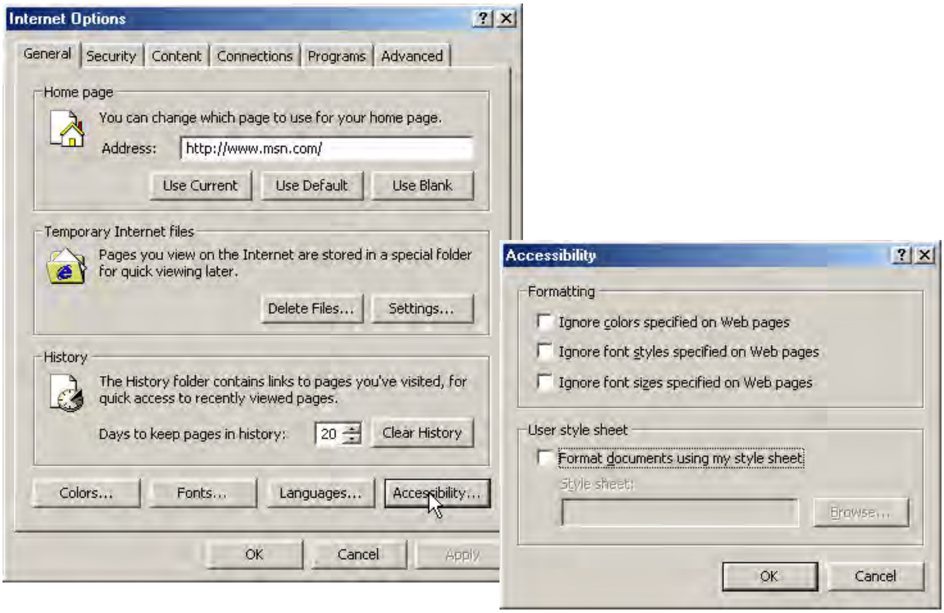

On-Screen Keyboard
File Keyboard Settings Help
esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 psc slk brk
漢字 1 2 3 4 5 6 7 8 9 0 - = bksp ins hm pup nlk / * -
tab q w e r t y u i o p [ ] ent del end pdn 7 8 9 +
英数/Lock a s d f g h j k l ; ' \ 4 5 6
shft z x c v b n m , . / shft 1 2 3 ent
ctrl alt 無変換 変換 かな alt ctrl 0 .

Fig. 24.42 Microsoft **On-Screen Keyboard**.

24.15.6 Accessibility Features in Microsoft Internet Explorer 5.5

Internet Explorer 5.5 offers a variety of options that can improve usability. To access IE5.5's accessibility features, launch the program, click the **Tools** menu and select **Internet Options...**. Then, from the **Internet Options** menu, press the button labeled **Accessibility...** to open the accessibility options (Fig. 24.43).

The accessibility options in IE5.5 are designed to improve the Web browsing experiences of users with disabilities. Users are able to ignore Web colors, Web fonts and font-size tags. This eliminates accessibility problems arising from poor Web-page design and allows users to customize their Web browsing. Users can even specify a *style sheet*, which formats every Web site that users visit according to their personal preferences.


Internet Options
General Security Content Connections Programs Advanced
Home page
Use Current Use Default Use Blank
Temporary Internet files
Delete Files... Settings...
History
Clear History
Colors... Fonts... Languages... Accessibility...
OK Cancel Apply
Accessibility
Formatting
User style sheet
Browse...
OK Cancel

You can change which page to use for your home page.

Address: http://www.msn.com/

Pages you view on the Internet are stored in a special folder for quick viewing later.

The History folder contains links to pages you've visited, for quick access to recently viewed pages.

Days to keep pages in history: 20

☐ Ignore colors specified on Web pages

☐ Ignore font styles specified on Web pages

☐ Ignore font sizes specified on Web pages

☐ Format documents using my style sheet

Style sheet:

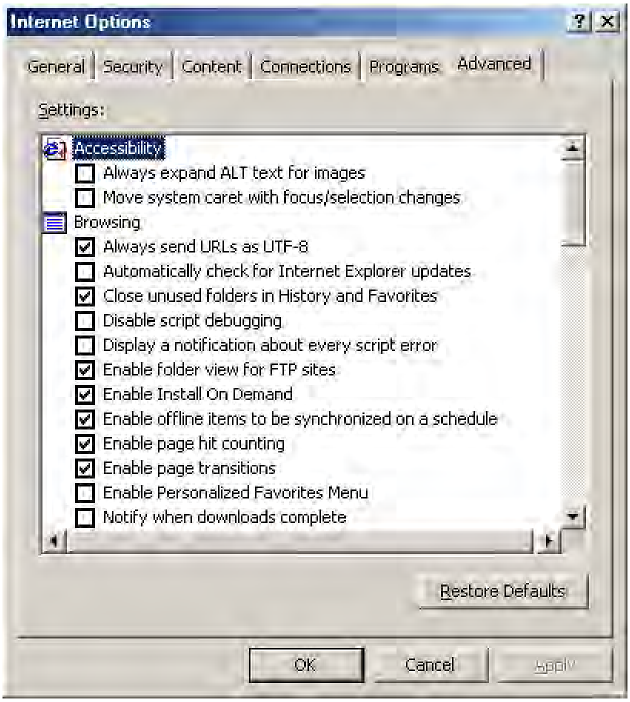
Fig. 24.43 Microsoft Internet Explorer 5.5's accessibility options.

In the **Internet Options** dialog, click the **Advanced** tab. This opens the dialog depicted in Fig. 24.44. The first available option is labeled **Always expand ALT text for images**. By default, IE5.5 hides some of the `<alt>` text if the size of the text exceeds that of the image it describes. This option forces IE5.5 to show all the text. The second option reads: **Move system caret with focus/selection changes**. This option is intended to make screen reading more effective. Some screen readers use the *system caret* (the blinking vertical bar associated with editing text) to determine what to read. If this option is not activated, screen readers might not read Web pages correctly.

Web designers often forget to take accessibility into account when creating Web sites, and, in attempts to provide large amounts of content, they use fonts that are too small. Many user agents have addressed this problem by allowing the user to adjust the text size. Click the **View** menu and select **Text Size** to change the font size in pages rendered by IE5.5. By default, the text size is set to **Medium**.

In this chapter, we presented a wide variety of technologies that help people with various disabilities use computers and the Internet. We hope that all our readers will join us in emphasizing the importance of these capabilities in their schools and workplaces.

Well, that's it for now. We sincerely hope that you have enjoyed learning with *C# How To Program*. As this book went to the presses, we were already at work on *Advanced C# How To Program*, a book appropriate for professional developers writing enterprise applications and for students enrolled in advanced software-development courses.



Accessibility
- ☐ Always expand ALT text for images
- ☐ Move system caret with focus/selection changes

Browsing
- ☑ Always send URLs as UTF-8
- ☐ Automatically check for Internet Explorer updates
- ☑ Close unused folders in History and Favorites
- ☐ Disable script debugging
- ☐ Display a notification about every script error
- ☑ Enable folder view for FTP sites
- ☑ Enable Install On Demand
- ☑ Enable offline items to be synchronized on a schedule
- ☑ Enable page hit counting
- ☑ Enable page transitions
- ☐ Enable Personalized Favorites Menu
- ☐ Notify when downloads complete

Fig. 24.44 Advanced accessibility settings in Microsoft Internet Explorer 5.5.

24.16 Internet and World Wide Web Resources

There are many accessibility resources available on the Internet and World Wide Web; this section lists a variety of these resources.

General Information, Guidelines and Definitions

`www.w3.org/WAI`
The World Wide Web Consortium's *Web Accessibility Initiative (WAI)* site promotes the design of universally accessible Web sites. This site contains the current guidelines and forthcoming standards for Web accessibility.

`www.w3.org/TR/xhtml1`
The *XHTML 1.0 Recommendation* contains XHTML 1.0 general information, compatibility issues, document type definition information, definitions, terminology and much more.

`www.abledata.com/text2/icg_hear.htm`
This page contains a consumer guide that discusses technologies designed for people with hearing impairments.

`www.washington.edu/doit`
The University of Washington's DO-IT (Disabilities, Opportunities, Internetworking and Technology) site provides information and Web-development resources for the creation of universally accessible Web sites.

`www.webable.com`
The *WebABLE* site contains links to many disability-related Internet resources; the site is geared towards those developing technologies for people with disabilities.

`www.webaim.org`
The *WebAIM* site provides a number of tutorials, articles, simulations and other useful resources that demonstrate how to design accessible Web sites. The site provides a screen-reader simulation.

`deafness.about.com/health/deafness/msubvib.htm`
This site provides information on vibrotactile devices, which allow individuals with hearing impairments to experience audio in the form of vibrations.

Developing Accessible Applications with Existing Technologies

`wdvl.com/Authoring/Languages/XML/XHTML`
The Web Developers Virtual Library provides an introduction to XHTML. This site also contains articles, examples and links to other technologies.

`www.w3.org/TR/1999/xhtml-modularization-19990406/DTD/doc`
The XHTML 1.0 DTD documentation site provides links to DTD documentation for the strict, transitional and frameset document type definitions.

`www.webreference.com/xml/reference/xhtml.html`
This Web page contains a list of the frequently used XHTML tags, such as header tags, table tags, frame tags and form tags. It also provides a description of each tag.

`www.w3.org/TR/REC-CSS2/aural.html`
This site discusses Aural Style Sheets, outlining the purpose and uses of this new technology.

`www.islandnet.com`
Lynxit is a development tool that allows users to view any Web site as if they were using a text-only browser. The site's form allows you to enter a URL and returns the Web site in text-only format.

`www.trill-home.com/lynx/public_lynx.html`
This site allows users to browse the Web with a Lynx browser. Users can view how Web pages appear to users who are not using the most current technologies.

`java.sun.com/products/java-media/speech/forDevelopers/JSML`
This site outlines the specifications for JSML, Sun Microsystem's Java Speech Markup Language. This language, like VoiceXML, helps improve accessibility for people with visual impairments.

`ocfo.ed.gov/coninfo/clibrary/software.htm`
This is the U.S. Department of Education's Web site that outlines software accessibility requirements. The site helps developers produce accessible products.

`www.speech.cs.cmu.edu/comp.speech/SpeechLinks.html`
The *Speech Technology Hyperlinks* page has over 500 links to sites related to computer-based speech and speech-recognition tools.

`www.islandnet.com/accessibility.html`
This page provides a list of tips for creating accessible Web pages.

`www.chantinc.com/technology`
This page is the *Chant* Web site, which discusses speech technology and how it works. Chant also provides speech–synthesis and speech-recognition software.

`searchmiddleware.techtarget.com/sdefinition/0,,sid26_gci518993,00.html`
This site provides definitions and information about several topics, including CallXML. Its thorough definition of CallXML differentiates CallXML from VoiceXML, another technology developed by Voxeo. The site also contains links to other published articles that discuss CallXML.

`www.oasis-open.org/cover/callxmlv2.html`
This site provides a comprehensive list of the CallXML tags, complete with a description of each tag. The site also provides short examples on how to apply the tags in various applications.

`web.ukonline.co.uk/ddmc/software.html`
This site provides links to software designed for people with disabilities.

`www.freedomscientific.com`
Henter-Joyce is a division of Freedom Scientific that provides software for people with visual impairments. It is the homepage of JAWS (Job Access with Sound).

`www-3.ibm.com/able/`
This is the homepage of IBM's accessibility site. It provides information on IBM products and their accessibility and discusses hardware, software and Web accessibility.

`www.w3.org/TR/voice-tts-reqs`
This page explains the speech-synthesis markup requirements for voice markup languages.

`www.cast.org`
CAST (Center for Applied Special Technology) offers software, including a valuable accessibility checker, that can help individuals with disabilities use computers. The accessibility checker is a Web-based program that validates the accessibility of Web sites.

Information on Disabilities

`deafness.about.com/health/deafness/msubmenu6.htm`
This is the home page of `deafness.about.com`. It provides a wealth of information on the history of hearing loss, the current state of medical developments and other resources related to these topics.

`www.trainingpost.org/3-2-inst.htm`
This site presents a tutorial on the Gunning Fog Index. The Gunning Fog Index is a method of grading text according to its readability.

`laurence.canlearn.ca/English/learn/accessibility2001/neads/index.shtml`
INDIE stands for "Integrated Network of Disability Information and Education." This site is home to a search engine that helps users find information on disabilities.

`www.wgbh.org/wgbh/pages/ncam/accesslinks.html`
This page provides links to other accessibility pages across the Web.

SUMMARY

- Enabling a Web site to meet the needs of individuals with disabilities is an important issue.
- Enabling a Web site to meet the needs of individuals with disabilities is an issue relevant to all business owners.
- Technologies such as voice activation, visual enhancers and auditory aids enable individuals with disabilities to have access to the web and software applications.
- In 1997, the World Wide Web Consortium (W3C) launched the Web Accessibility Initiative (WAI). The WAI is an attempt to make the Web more accessible; its mission is described at **`www.w3.org/WAI`**.
- Accessibility refers to the level of usability of an application or Web site for people with disabilities. Total accessibility is difficult to achieve because there are many different disabilities, language barriers, and hardware and software inconsistencies.
- The majority of Web sites are considered to be either partially or totally inaccessible to people with visual, learning or mobility impairments.
- The WAI published the Web Content Accessibility Guidelines 1.0, which assign accessibility priorities to a three-tier structure of checkpoints. The WAI currently is working on a draft of the Web Content Accessibility Guidelines 2.0.
- One important WAI requirement is to ensure that every image, movie and sound on a Web site is accompanied by a description that clearly defines the item's purpose; the description is called an **`<alt>`** tag.
- Specialized user agents, such as screen readers (programs that allow users to hear what is being displayed on their screen) and braille displays (devices that receive data from screen-reading software and output the data as braille), allow people with visual impairments to access text-based information that normally is displayed on the screen.
- Using a screen reader to navigate a Web site can be time consuming and frustrating, because screen readers are unable to interpret pictures and other graphical content that do not have alternative text.
- Including links at the top of each Web page provides easy access to the page's main content.
- Web pages with large amounts of multimedia content are difficult for user agents to interpret unless they are designed properly. Images, movies and most non-XHTML objects cannot be read by screen readers.
- Misused heading tags (**`<h1>`**) also present challenges to some Web users—particularly those who cannot use a mouse.
- Web designers should avoid misuse of the **`alt`** attribute; it is intended to provide a short description of an XHTML object that might not load properly on all user agents.

- The value of the **`longdesc`** attribute is a text-based URL, linked to a Web page, that describes the image associated with the attribute.
- When creating a Web page for the general public, it is important to consider the reading level at which it is written. Web site designers can make their sites more readable through the use of shorter words; some users may have difficulty understanding slang and other nontraditional language.
- Web designers often use frames to display more than one XHTML file at a time. Unfortunately, frames often lack proper descriptions, which prevents users with text-based browsers and users with visual impairments from navigating the Web site.
- The **`<noframes>`** tag allows the designer to offer alternative content to users whose browsers do not support frames.
- VoiceXML has tremendous implications for people with visual impairments and for illiterate people. VoiceXML, a speech recognition and synthesis technology, reads Web pages to users and understands words spoken into a microphone.
- A VoiceXML document is composed of a series of dialogs and subdialogs, which result in spoken interaction between the user and the computer. VoiceXML is a voice-recognition technology.
- CallXML, a language created and supported by Voxeo, creates phone-to-Web applications. These applications tailor themselves to the user's input.
- When a user accesses a CallXML application, the incoming telephone call is referred to as a session. A CallXML application can support multiple sessions that enable the application to receive multiple telephone calls at any given time.
- A session terminates either when the user hangs up the telephone or when the CallXML application invokes the **`hangup`** element.
- The contents of a CallXML application are inserted within the **`<callxml>`** tag.
- CallXML tags that perform similar tasks should be enclosed between the **`<block>`** and **`</block>`** tags.
- To deploy a CallXML application, register with the Voxeo Community, which assigns a telephone number to the application so that other users may access it.
- Voxeo's logging feature enables developers to debug their telephone application by observing the "conversation" between the user and the application.
- Braille keyboards are similar to standard keyboards, except that in addition to having each key labeled with the letter it represents, braille keyboards have the equivalent braille symbol printed on the key. Most often, braille keyboards are combined with a speech synthesizer or a braille display, so users are able to interact with the computer to verify that their typing is correct.
- People with visual impairments are not the only beneficiaries of the effort being made to improve markup languages. Individuals with hearing impairments also have a great number of tools to help them interpret auditory information delivered over the Web.
- Speech synthesis is another area in which research is being done to help people with disabilities.
- Open-source software for people with visual impairments already exists and is often superior to most of its proprietary, closed-source counterparts. However, it still does not use the Linux OS to its fullest extent.
- People with hearing impairments will soon benefit from what is called Synchronized Multimedia Integration Language (SMIL). This markup language is designed to add extra tracks—layers of content found within a single audio or video file. The additional tracks can contain such data as closed captioning.

- EagleEyes, developed by researchers at Boston College (`www.bc.edu/eagleeyes`), is a system that translates eye movements into mouse movements. Users move the mouse cursor by moving their eyes or head and are thereby able to control the computer.
- All of the accessibility options provided by Windows 2000 are available through the **Accessibility Wizard**. The **Accessibility Wizard** takes a user step by step through all of the Windows accessibility features and configures his or her computer according to the chosen specifications.
- Microsoft Magnifier enlarges the section of your screen surrounding the mouse cursor.
- To solve problems seeing the mouse cursor, Microsoft offers the ability to use larger cursors, black cursors and cursors that invert objects underneath them.
- **SoundSentry** is a tool that creates visual signals when system events occur.
- **ShowSounds** adds captions to spoken text and other sounds produced by today's multimedia-rich software.
- **StickyKeys** is a program that helps users who have difficulty pressing multiple keys at the same time.
- **BounceKeys** forces the computer to ignore repeated keystrokes, solving the problem of accidentally pressing the same key more than once.
- **ToggleKeys** causes an audible beep to alert users that they have pressed one of the lock keys (i.e., *Caps Lock*, *Num Lock*, or *Scroll Lock*).
- **MouseKeys** is a tool that uses the keyboard to emulate mouse movements.
- The **Mouse Button Settings** tool allows you to create a virtual left-handed mouse by swapping the button functions.
- A timeout either enables or disables a certain action after the computer has idled for a specified amount of time. A common use of a timeout is in a screen saver.
- Default settings are loaded when the computer is rebooted.
- You can create an `.acw` file, which, when chosen, will automatically activate the saved accessibility settings on any Windows 2000 computer.
- Microsoft **Narrator** is a text-to-speech program for people with visual impairments. It reads text, describes the current desktop environment and alerts the user when certain Windows events occur.

TERMINOLOGY

`<alt>` tag
accessibility
accessibility aids in Visual Studio .NET
Accessibility Wizard
Accessibility Wizard initialization option
Accessibility Wizard mouse-cursor adjustment tool
`AccessibilityDescription` property of class `Control`
`AccessibilityName` property of class `Control`
`AccessibleDescription` property of class `Control`
`AccessibleName` property of class `Control`
`AccessibleRole` enumeration
`AccessibleRole` property of class `Control`
action element
Active Accessibility
Acts designed to ensure Internet access for people with disabilities
`.acw`
ADA (Americans with Disabilities Act)
advanced accessibility settings in Microsoft Internet Explorer 5.5
`alt` attribute
Americans with Disabilities Act (ADA)
`answer` element
`<assign>` tag (`<assign>…</assign>`)
`assign` element
Aural Style Sheet
AuralCSS
`block` element

`<block>` tag (`<block>…</block>`)
BounceKeys
braille display
braille keyboard
`<break>` tag (`<break>…</break>`)
`call` element
`callerID` attribute
CallXML
`callxml` element
CallXML elements
CallXML `hangup` element
`caption` element
Cascading Style Sheets (CSS)
CAST eReader
Center for Applied Special Technology
`choice` element of `form` tag
`choice` element of `menu` tag
`<choice>` tag (`<choice>…</choice>`)
`clear` element
`clearDigits` element
Clicker 4
`conference` element
CORDA Technologies
`count` attribute if `prompt` element
CSS (Cascading Style Sheets)
CSS2
default setting
Display Color Settings
Display Settings
D-link
EagleEyes
Emacspeak
`encoding` declaration
end of session message
`<enumerate>` tag (`<enumerate>…</enumerate>`)
event handler
exam
`hello.xml`
`isbn.xml`
`main.vxml`
`publications.vxml`
`withheaders.html`
`withoutheaders.html`
`<exit>` tag (`<exit>…</exit>`)
Extra Keyboard Help
`<filled>` tag (`<filled>…</filled>`)
Font Size dialog
`<form>` tag (`<form>…</form>`)
`format` attribute
frame
Freedom Scientific
get request type
`getDigits` element
global variable
`goto` element
`<goto>` tag (`<goto>…</goto>`)
`<grammar>` tag (`<grammar>…</grammar>`)
Gunning Fog Index
`headers` attribute
Henter-Joyce
Home Page Reader (HPR)
HPR (Home Page Reader)
HTTP (HyperText Transfer Protocol)
`<if>` tag (`<if>…</if>`)
`img` element
inclusive technology
`<input>`
`IsAccessible` property of class `Control`
Java Development Kit (Java SDK 1.3)
JAWS (Job Access with Sound)
JSML
linearized
`link` element in VoiceXML
`<link>` tag (`<link>…</link>`)
local dialog
logging feature
logic element
`longdesc` attribute
Lynx
`maxDigits` attribute
`maxTime` attribute
`<menu>` tag (`<menu>…</menu>`)
`method` attribute
Microsoft Internet Explorer accessibility options
Microsoft **Magnifier**
Microsoft **Narrator**
Microsoft **On-Screen Keyboard**
Mouse Button Settings
mouse cursor
Mouse Speed dialog
`MouseHover` event
MouseKeys
Narrator reading **Notepad** text
`next` attribute of `choice` element
`object`
Ocularis
`onHangup` element
`onMaxSilence` element
On-Screen Keyboard

onTermDigit element
play element
post request type
prompt element in VoiceXML
<prompt> tag (**<prompt>…</prompt>**)
RDK (Redistribution Kit)
readability
recordAudio element
Redistribution Kit (RDK)
run element
screen reader
scroll bar and window border size dialog
SDK (Software Development Kit)
sendEvent element
session
session attribute
sessionID
Set Automatic Timeouts
setting up window element size
shortcut key
ShowSounds
SMIL (Synchronized Multimedia Integration Language)
Software Development Kit (SDK)
SoundSentry
speech recognition
speech synthesis
speech synthesizer
StickyKeys
style sheet
<subdialog> tag (**<subdialog>…</subdialog>**)
submit attribute
summary attribute
Synchronized Multimedia Integration Language (SMIL)
system caret
tab order
tab stop
TabIndex property of class **Control**
table
TabStop property of class **Control**
targetSessions attribute
termDigits attribute
text element
text to speech (TTS)
th element
timeout
timeout attribute of **prompt** element
title tag (**<title>…</title>**)
ToggleKeys
track
TTS (text-to-speech) engine
Type class
user agent
value attribute
<var> tag (**<var>…</var>**)
var attribute
version declaration
ViaVoice
Visual Studio accessibility guidelines
Voice Server SDK 1.0
voice synthesis
voice technology
VoiceXML
VoiceXML tags
Voxeo (**www.voxeo.com**)
Voxeo Account Manager
<vxml> tag (**<vxml>…</vxml>**)
WAI (Web Accessibility Initiative)
WAI Quick Tip
wait element
Web Accessibility Initiative (WAI)
Web Content Accessibility Guidelines 1.0
Web Content Accessibility Guidelines 2.0 (Working Draft)
World Wide Web Consortium (W3C)
www.voxeo.com (Voxeo)
XHTML Recommendation
XML GL (XML Guidelines)
XML Guidelines (XML GL)

SELF-REVIEW EXERCISES

24.1 Expand the following acronyms:
a) W3C.
b) WAI.
c) JAWS.
d) SMIL.
e) CSS.

24.2 Fill in the blanks in each of the following statements.

a) The highest priority of the Web Accessibility Initiative is to ensure that __________, __________ and __________ are accompanied by descriptions that clearly define their purposes.

b) Technologies such as __________, __________ and __________ enable individuals with disabilities to work in a large number of positions.

c) Although they are a great layout tool for presenting data, __________ are difficult for screen readers to interpret and convey clearly to a user.

d) To make a frame accessible to individuals with disabilities, it is important to include __________tags on the page.

e) Blind people using computers often are assisted by __________ and __________.

f) CallXML is used to create __________ applications that allow individuals to receive and send telephone calls.

g) A __________ tag must be associated with the `<getDigits>` tag.

24.3 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Screen readers have no problem reading and translating images.

b) When writing Web pages for the general public, it is important to consider the reading level of the context.

c) The `<alt>` tag helps screen readers describe the images on a Web page.

d) Blind people have been helped by the improvements made in speech-recognition technology more than any other group of people.

e) VoiceXML lets users interact with Web content using speech recognition and speech synthesis technologies.

f) Elements such as `onMaxSilence`, `onTermDigit` and `onMaxTime` are event handlers because they perform specified tasks when invoked.

g) The debugging feature of the **Voxeo Account Manager** assists developers in debugging their CallXML applications.

ANSWERS TO SELF-REVIEW EXERCISES

24.1 a) World Wide Web Consortium. b) Web Accessibility Initiative. c) Job Access with Sound. d) Synchronized Multimedia Integration Language. e) Cascading Style Sheets.

24.2 a) image, movie, sound. b) voice activation, visual enhancers and auditory aids. c) tables. d) `<noframes>`. e) braille displays, braille keyboards. f) phone-to-Web. g) `<onTermDigit>`.

24.3 a) False. Screen readers cannot directly interpret images. If the programmer includes an `alt` attribute inside the `<img>` tag, the screen reader reads this description to the user. b) True. c) True. d) False. Although speech-recognition technology has had a large impact on blind people, speech-recognition technology has had also a large impact on people who have trouble typing. e) True. f) True. g) False. The logging feature assists developers in debugging their CallXML application.

EXERCISES

24.4 Insert XHTML markup into each segment to make the segment accessible to someone with disabilities. The contents of images and frames should be apparent from the context and filenames.

a) `<img src = "dogs.jpg" width = "300" height = "250" />`

b)
```
<table width = "75%">
   <tr><th>Language</th><th>Version</th></tr>
   <tr><td>XHTML</td><td>1.0</td></tr>
```

```
      <tr><td>Perl</td><td>5.6.0</td></tr>
      <tr><td>Java</td><td>1.3</td></tr>
   </table>
```

c)
```
   <map name = "links">
      <area href = "index.html" shape = "rect"
         coords = "50, 120, 80, 150" />
      <area href = "catalog.html" shape = "circle"
         coords = "220, 30" />
   </map>
   <img src = "antlinks.gif" width = "300" height = "200"
      usemap = "#links" />
```

24.5 Define the following terms:

a) Action element.
b) Gunning Fog Index.
c) Screen reader.
d) Session.
e) Web Accessibility Initiative (WAI).

24.6 Describe the three-tier structure of checkpoints (priority-one, priority-two and priority-three) set forth by the WAI.

24.7 Why do misused `<h1>` heading tags create problems for screen readers?

24.8 Use CallXML to create a voice-mail system that plays a voice-mail greeting and records a message. Have friends and classmates call your application and leave a message.

A

Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom with each level of precedence separated by a horizontal line.[1]

Operator	Type	Associativity
`.`	member access	left-to-right
`()`	parenthesized expression	
`[]`	element access	
`++`	post increment	
`--`	post decrement	
`new`	object creation	
`typeof`	typeof	
`checked`	checked	
`unchecked`	unchecked	
`+`	unary plus	left-to-right
`-`	unary minus	
`!`	unary	
`~`	unary	
`++`	pre-increment	
`--`	pre-decrement	

Fig. A.1 Operator precedence chart. (Part 1 of 2.)

---

[1] This operator-precedence chart is based on Section 7.2.1, *Operator precedence and associativity*, of the C# Language Specification (for more information, visit **`msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/CSharpSpecStart.asp`**).

Operator	Type	Associativity
`*`	multiplication	left-to-right
`/`	division	
`%`	modulus	
`+`	addition	left-to-right
`-`	subtraction	
`<<`	shift left	left-to-right
`>>`	shift right	
`<`	relational less than	left-to-right
`>`	relational greater than	
`<=`	relational less than or equal to	
`>=`	relational greater than or equal to	
`is`	type comparison	
`==`	relational is equal to	left-to-right
`!=`	relational is not equal to	
`&`	logical AND	left-to-right
`^`	logical exclusive OR	left-to-right
`|`	logical inclusive OR	left-to-right
`&&`	conditional AND	left-to-right
`||`	conditional OR	left-to-right
`?:`	conditional	right-to-left
`=`	assignment	right-to-left
`*=`	multiplication assignment	
`/=`	division assignment	
`+=`	addition assignment	
`-=`	subtraction assignment	
`<<=`	shift left assignment	
`>>=`	shift right assignment	
`&=`	logical AND assignment	
`^=`	logical exclusive OR assignment	
`|=`	logical inclusive OR assignment	

Fig. A.1 Operator precedence chart. (Part 2 of 2.)

B

Number Systems

Objectives

- To understand basic number system concepts such as base, positional value and symbol value.
- To understand how to work with numbers represented in the binary, octal and hexadecimal number systems
- To be able to abbreviate binary numbers as octal numbers or hexadecimal numbers.
- To be able to convert octal numbers and hexadecimal numbers to binary numbers.
- To be able to covert back and forth between decimal numbers and their binary, octal and hexadecimal equivalents.
- To understand binary arithmetic and how negative binary numbers are represented using two's complement notation.

Here are only numbers ratified.
William Shakespeare

Nature has some sort of arithmetic-geometrical coordinate system, because nature has all kinds of models. What we experience of nature is in models, and all of nature's models are so beautiful.
It struck me that nature's system must be a real beauty, because in chemistry we find that the associations are always in beautiful whole numbers—there are no fractions.
Richard Buckminster Fuller

Outline

B.1 Introduction

B.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers

B.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers

B.4 Converting from Binary, Octal or Hexadecimal to Decimal

B.5 Converting from Decimal to Binary, Octal, or Hexadecimal

B.6 Negative Binary Numbers: Two's Complement Notation

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

B.1 Introduction

In this appendix, we introduce the key number systems that programmers use, especially when they are working on software projects that require close interaction with "machine-level" hardware. Projects like this include operating systems, computer networking software, compilers, database systems, and applications requiring high performance.

When we write an integer such as 227 or –63 in a program, the number is assumed to be in the *decimal (base 10) number system*. The *digits* in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The lowest digit is 0 and the highest digit is 9—one less than the *base* of 10. Internally, computers use the *binary (base 2) number system*. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2. Fig. B.1 summarizes the digits used in the binary, octal, decimal and hexadecimal number systems.

As we will see, binary numbers tend to be much longer than their decimal equivalents. Programmers who work in assembly languages and in high-level languages that enable programmers to reach down to the "machine level," find it cumbersome to work with binary numbers. So two other number systems the *octal number system (base 8)* and the *hexadecimal number system (base 16)*—are popular primarily because they make it convenient to abbreviate binary numbers.

In the octal number system, the digits range from 0 to 7. Because both the binary number system and the octal number system have fewer digits than the decimal number system, their digits are the same as the corresponding digits in decimal.

The hexadecimal number system poses a problem because it requires sixteen digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15. Thus in hexadecimal we can have numbers like 876 consisting solely of decimal-like digits, numbers like 8A55F consisting of digits and letters, and numbers like FFE consisting solely of letters. Occasionally, a hexadecimal number spells a common word such as FACE or FEED—this can appear strange to programmers accustomed to working with numbers. Fig. B.2 summarizes each of the number systems.

Each of these number systems uses positional notation—each position in which a digit is written has a different positional value. For example, in the decimal number 937 (the 9, the 3, and the 7 are referred to as symbol values), we say that the 7 is written in the ones position, the 3 is written in the tens position, and the 9 is written in the hundreds position. Notice that each of these positions is a power of the base (base 10), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. B.3).

For longer decimal numbers, the next positions to the left would be the thousands position (10 to the 3rd power), the ten-thousands position (10 to the 4th power), the hundred-thousands position (10 to the 5th power), the millions position (10 to the 6th power), the ten-millions position (10 to the 7th power) and so on.

In the binary number 101, we say that the rightmost 1 is written in the ones position, the 0 is written in the twos position, and the leftmost 1 is written in the fours position. Notice that each of these positions is a power of the base (base 2), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. B.4).

For longer binary numbers, the next positions to the left would be the eights position (2 to the 3rd power), the sixteens position (2 to the 4th power), the thirty-twos position (2 to the 5th power), the sixty-fours position (2 to the 6th power), and so on.

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position, and the 4 is written in the sixty-fours position. Notice that each of these positions is a power of the base (base 8), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. B.5).

Binary digit	**Octal digit**	**Decimal digit**	**Hexadecimal digit**
`0`	`0`	`0`	`0`
`1`	`1`	`1`	`1`
	`2`	`2`	`2`
	`3`	`3`	`3`
	`4`	`4`	`4`
	`5`	`5`	`5`
	`6`	`6`	`6`
	`7`	`7`	`7`
		`8`	`8`
		`9`	`9`
			`A` (decimal value of 10)
			`B` (decimal value of 11)
			`C` (decimal value of 12)
			`D` (decimal value of 13)
			`E` (decimal value of 14)
			`F` (decimal value of 15)

Fig. B.1 Digits of the binary, octal, decimal and hexadecimal number systems.

Attribute	Binary	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Lowest digit	0	0	0	0
Highest digit	1	7	9	F

Fig. B.2 Comparison of the binary, octal, decimal and hexadecimal number systems.

Positional values in the decimal number system			
Decimal digit	9	3	7
Position name	Hundreds	Tens	Ones
Positional value	100	10	1
Positional value as a power of the base (10)	10^2	10^1	10^0

Fig. B.3 Positional values in the decimal number system.

For longer octal numbers, the next positions to the left would be the five-hundred-and-twelves position (8 to the 3rd power), the four-thousand-and-ninety-sixes position (8 to the 4th power), the thirty-two-thousand-seven-hundred-and-sixty eights position (8 to the 5th power), and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position, and the 3 is written in the two-hundred-and-fifty-sixes position. Notice that each of these positions is a power of the base (base 16), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. B.6).

For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16 to the 3rd power), the sixty-five-thousand-five-hundred-and-thirty-six position (16 to the 4th power), and so on.

Positional values in the binary number system			
Binary digit	1	0	1
Position name	Fours	Twos	Ones
Positional value	4	2	1
Positional value as a power of the base (2)	2^2	2^1	2^0

Fig. B.4 Positional values in the binary number system.

Positional values in the octal number system			
Decimal digit	`4`	`2`	`5`
Position name	Sixty-fours	Eights	Ones
Positional value	`64`	`8`	`1`
Positional value as a power of the base (8)	8^2	8^1	8^0

Fig. B.5 Positional values in the octal number system.

Positional values in the hexadecimal number system			
Decimal digit	`3`	`D`	`A`
Position name	Two-hundred-and-fifty-sixes	Sixteens	Ones
Positional value	`256`	`16`	`1`
Positional value as a power of the base (16)	16^2	16^1	16^0

Fig. B.6 Positional values in the hexadecimal number system.

B.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers

The main use for octal and hexadecimal numbers in computing is for abbreviating lengthy binary representations. Figure B.7 highlights the fact that lengthy binary numbers can be expressed concisely in number systems with higher bases than the binary number system.

Decimal number	Binary representation	Octal representation	Hexadecimal representation
`0`	`0`	`0`	`0`
`1`	`1`	`1`	`1`
`2`	`10`	`2`	`2`
`3`	`11`	`3`	`3`
`4`	`100`	`4`	`4`
`5`	`101`	`5`	`5`
`6`	`110`	`6`	`6`
`7`	`111`	`7`	`7`

Fig. B.7 Decimal, binary, octal, and hexadecimal equivalents (part 1 of 2).

Decimal number	Binary representation	Octal representation	Hexadecimal representation
`8`	`1000`	`10`	`8`
`9`	`1001`	`11`	`9`
`10`	`1010`	`12`	`A`
`11`	`1011`	`13`	`B`
`12`	`1100`	`14`	`C`
`13`	`1101`	`15`	`D`
`14`	`1110`	`16`	`E`
`15`	`1111`	`17`	`F`
`16`	`10000`	`20`	`10`

Fig. B.7 Decimal, binary, octal, and hexadecimal equivalents (part 2 of 2).

A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2). Consider the following 12-digit binary number and its octal and hexadecimal equivalents. See if you can determine how this relationship makes it convenient to abbreviate binary numbers in octal or hexadecimal. The answer follows the numbers.

Binary Number	Octal equivalent	Hexadecimal equivalent
`100011010001`	`4321`	`8D1`

To see how the binary number converts easily to octal, simply break the 12-digit binary number into groups of three consecutive bits each, and write those groups over the corresponding digits of the octal number as follows

```
100     011     010     001
4       3       2       1
```

Notice that the octal digit you have written under each group of thee bits corresponds precisely to the octal equivalent of that 3-digit binary number as shown in Fig. B.7.

The same kind of relationship may be observed in converting numbers from binary to hexadecimal. In particular, break the 12-digit binary number into groups of four consecutive bits each and write those groups over the corresponding digits of the hexadecimal number as follows

```
1000    1101    0001
8       D       1
```

Notice that the hexadecimal digit you wrote under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number as shown in Fig. B.7.

B.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting these groups as their equivalent octal digit values or hexadecimal digit values. This process may be used in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101, and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101, and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

B.4 Converting from Binary, Octal or Hexadecimal to Decimal

Because we are accustomed to working in decimal, it is often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is "really" worth. Our diagrams in Section B.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value, and sum these products. For example, the binary number 110101 is converted to decimal 53 as shown in Fig. B.8.

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values as shown in Fig. B.9.

Converting a binary number to decimal

Positional values:	`32`	`16`	`8`	`4`	`2`	`1`
Symbol values:	`1`	`1`	`0`	`1`	`0`	`1`
Products:	`1*32=32`	`1*16=16`	`0*8=0`	`1*4=4`	`0*2=0`	`1*1=1`
Sum:	`= 32 + 16 + 0 + 4 + 0 + 1 = 53`					

Fig. B.8 Converting a binary number to decimal.

Converting an octal number to decimal

Positional values:	`512`	`64`	`8`	`1`
Symbol values:	`7`	`6`	`1`	`4`
Products	`7*512=3584`	`6*64=384`	`1*8=8`	`4*1=4`
Sum:	`= 3584 + 384 + 8 + 4 = 3980`			

Fig. B.9 Converting an octal number to decimal.

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values as shown in Fig. B.10.

B.5 Converting from Decimal to Binary, Octal, or Hexadecimal

The conversions of the previous section follow naturally from the positional notation conventions. Converting from decimal to binary, octal or hexadecimal also follows these conventions.

Suppose we wish to convert decimal 57 to binary. We begin by writing the positional values of the columns right to left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	`64`	`32`	`16`	`8`	`4`	`2`	`1`

Then we discard the column with positional value 64 leaving:

Positional values:	`32`	`16`	`8`	`4`	`2`	`1`

Next we work from the leftmost column to the right. We divide 32 into 57 and observe that there is one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. We divide 16 into 25 and observe that there is one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We divide 8 into 9 and observe that there is one 8 in 9 with a remainder of 1. The next two columns each produce quotients of zero when their positional values are divided into 1 so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1 so we write 1 in the 1 column. This yields:

Positional values:	`32`	`16`	`8`	`4`	`2`	`1`
Symbol values:	`1`	`1`	`1`	`0`	`0`	`1`

and thus decimal 57 is equivalent to binary 111001.

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	`512`	`64`	`8`	`1`

Then we discard the column with positional value 512, yielding:

Positional values:	`64`	`8`	`1`

Converting a hexadecimal number to decimal

Positional values:	`4096`	`256`	`16`	`1`
Symbol values:	`A`	`D`	`3`	`B`
Products	`A*4096=40960`	`D*256=3328`	`3*16=48`	`B*1=11`
Sum:	`= 40960 + 3328 + 48 + 11 = 44347`			

Fig. B.10 Converting a hexadecimal number to decimal.

Next we work from the leftmost column to the right. We divide 64 into 103 and observe that there is one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe that there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder so we write 7 in the 1 column. This yields:

Positional values:	`64`	`8`	`1`
Symbol values:	`1`	`4`	`7`

and thus decimal 103 is equivalent to octal 147.

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write

Positional values:	`4096`	`256`	`16`	`1`

Then we discard the column with positional value 4096, yielding:

Positional values:	`256`	`16`	`1`

Next we work from the leftmost column to the right. We divide 256 into 375 and observe that there is one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe that there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder so we write 7 in the 1 column. This yields:

Positional values:	`256`	`16`	`1`
Symbol values:	`1`	`7`	`7`

and thus decimal 375 is equivalent to hexadecimal 177.

B.6 Negative Binary Numbers: Two's Complement Notation

The discussion in this appendix has been focussed on positive numbers. In this section, we explain how computers represent negative numbers using *two's complement notation*. First we explain how the two's complement of a binary number is formed, and then we show why it represents the negative value of the given binary number.

Consider a machine with 32-bit integers. Suppose

```
int number = 13;
```

The 32-bit representation of **`number`** is

```
00000000 00000000 00000000 00001101
```

To form the negative of **`number`** we first form its *one's complement* by applying C#'s ^ operator:

```
onesComplement = number ^ 0x7FFFFFFF;
```

Internally, **`onesComplement`** is now **`number`** with each of its bits reversed—ones become zeros and zeros become ones as follows:

```
number:
00000000 00000000 00000000 00001101

onesComplement:
11111111 11111111 11111111 11110010
```

To form the two's complement of **`number`** we simply add one to **`number`** one's complement. Thus

```
Two's complement of number:
11111111 11111111 11111111 11110011
```

Now if this is in fact equal to –13, we should be able to add it to binary 13 and obtain a result of 0. Let us try this:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
------------------------------------
 00000000 00000000 00000000 00000000
```

The carry bit coming out of the leftmost column is discarded and we indeed get zero as a result. If we add the one's complement of a number to the number, the result would be all 1s. The key to getting a result of all zeros is that the twos complement is 1 more than the one's complement. The addition of 1 causes each column to add to 0 with a carry of 1. The carry keeps moving leftward until it is discarded from the leftmost bit, and hence the resulting number is all zeros.

Computers actually perform a subtraction such as

```
x = a - number;
```

by adding the two's complement of **`number`** to **`a`** as follows:

```
x = a + ( onesComplement + 1 );
```

Suppose **`a`** is 27 and **`number`** is 13 as before. If the two's complement of **`number`** is actually the negative of **`number`**, then adding the two's complement of value to **`a`** should produce the result 14. Let us try this:

```
a (i.e., 27)                  00000000 00000000 00000000 00011011
+( onesComplement + 1 )      +11111111 11111111 11111111 11110011
                             ------------------------------------
                              00000000 00000000 00000000 00001110
```

which is indeed equal to 14.

SUMMARY

- When we write an integer such as 19 or 227 or –63 in a C# program, the number is automatically assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10.
- Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.

- The octal number system (base 8) and the hexadecimal number system (base 16) are popular primarily because they make it convenient to abbreviate binary numbers.
- The digits of the octal number system range from 0 to 7.
- The hexadecimal number system poses a problem because it requires sixteen digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15.
- Each number system uses positional notation—each position in which a digit is written has a different positional value.
- A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2).
- To convert an octal number to a binary number, simply replace each octal digit with its three-digit binary equivalent.
- To convert a hexadecimal number to a binary number, simply replace each hexadecimal digit with its four-digit binary equivalent.
- Because we are accustomed to working in decimal, it is convenient to convert a binary, octal or hexadecimal number to decimal to get a sense of the number's "real" worth.
- To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value, and sum these products.
- Computers represent negative numbers using two's complement notation.
- To form the negative of a value in binary, first form its one's complement by applying Visual Basic's `Xor` operator. This reverses the bits of the value. To form the two's complement of a value, simply add one to the value's one's complement.

TERMINOLOGY

base
base 2 number system
base 8 number system
base 10 number system
base 16 number system
binary number system
bitwise complement operator (~)
conversions
decimal number system
digit
hexadecimal number system
negative value
octal number system
one's complement notation
positional notation
positional value
symbol value
two's complement notation

SELF-REVIEW EXERCISES

B.1 The bases of the decimal, binary, octal, and hexadecimal number systems are __________, __________, __________, and __________ respectively.

B.2 In general, the decimal, octal, and hexadecimal representations of a given binary number contain (more/fewer) digits than the binary number contains.

B.3 (True/False) A popular reason for using the decimal number system is that it forms a convenient notation for abbreviating binary numbers simply by substituting one decimal digit per group of four binary bits.

B.4 The (octal / hexadecimal / decimal) representation of a large binary value is the most concise (of the given alternatives).

B.5 (True/False) The highest digit in any base is one more than the base.

B.6 (True/False) The lowest digit in any base is one less than the base.

B.7 The positional value of the rightmost digit of any number in either binary, octal, decimal, or hexadecimal is always __________.

B.8 The positional value of the digit to the left of the rightmost digit of any number in binary, octal, decimal, or hexadecimal is always equal to __________.

B.9 Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

decimal	`1000`	`100`	`10`	`1`
hexadecimal	`...`	`256`	`...`	`...`
binary	`...`	`...`	`...`	`...`
octal	`512`	`...`	`8`	`...`

B.10 Convert binary 110101011000 to octal and to hexadecimal.

B.11 Convert hexadecimal FACE to binary.

B.12 Convert octal 7316 to binary.

B.13 Convert hexadecimal 4FEC to octal. (Hint: First convert 4FEC to binary then convert that binary number to octal.)

B.14 Convert binary 1101110 to decimal.

B.15 Convert octal 317 to decimal.

B.16 Convert hexadecimal EFD4 to decimal.

B.17 Convert decimal 177 to binary, to octal, and to hexadecimal.

B.18 Show the binary representation of decimal 417. Then show the one's complement of 417, and the two's complement of 417.

B.19 What is the result when the one's complement of a number is added to itself?

SELF-REVIEW ANSWERS

B.1 10, 2, 8, 16.

B.2 Fewer.

B.3 False.

B.4 Hexadecimal.

B.5 False. The highest digit in any base is one less than the base.

B.6 False. The lowest digit in any base is zero.

B.7 1 (the base raised to the zero power).

B.8 The base of the number system.

B.9 Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

decimal	`1000`	`100`	`10`	`1`
hexadecimal	`4096`	`256`	`16`	`1`
binary	`8`	`4`	`2`	`1`
octal	`512`	`64`	`8`	`1`

B.10 Octal 6530; Hexadecimal D58.

B.11 Binary 1111 1010 1100 1110.

B.12 Binary 111 011 001 110.

B.13 Binary 0 100 111 111 101 100; Octal 47754.

B.14 Decimal 2+4+8+32+64=110.

B.15 Decimal 7+1*8+3*64=7+8+192=207.

B.16 Decimal 4+13*16+15*256+14*4096=61396.

B.17 Decimal 177
to binary:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

to octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

to hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

B.18 Binary:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+
(1*1)
110100001
```

```
One's complement: 001011110
Two's complement: 001011111
Check: Original binary number + its two's complement
```

```
110100001
001011111
---------
000000000
```

B.19 Zero.

EXERCISES

B.20 Some people argue that many of our calculations would be easier in the base 12 number system because 12 is divisible by so many more numbers than 10 (for base 10). What is the lowest digit in base 12? What might the highest symbol for the digit in base 12 be? What are the positional values of the rightmost four positions of any number in the base 12 number system?

B.21 How is the highest symbol value in the number systems we discussed related to the positional value of the first digit to the left of the rightmost digit of any number in these number systems?

B.22 Complete the following chart of positional values for the rightmost four positions in each of the indicated number systems:

`decimal`	`1000`	`100`	`10`	`1`
`base 6`	`...`	`...`	`6`	`...`
`base 13`	`...`	`169`	`...`	`...`
`base 3`	`27`	`...`	`...`	`...`

B.23 Convert binary 100101111010 to octal and to hexadecimal.

B.24 Convert hexadecimal 3A7D to binary.

B.25 Convert hexadecimal 765F to octal. (Hint: First convert 765F to binary, then convert that binary number to octal.)

B.26 Convert binary 1011110 to decimal.

B.27 Convert octal 426 to decimal.

B.28 Convert hexadecimal FFFF to decimal.

B.29 Convert decimal 299 to binary, to octal, and to hexadecimal.

B.30 Show the binary representation of decimal 779. Then show the one’s complement of 779, and the two’s complement of 779.

B.31 What is the result when the two’s complement of a number is added to itself?

B.32 Show the two’s complement of integer value –1 on a machine with 32-bit integers.

C

Career Opportunities

Objectives

- To explore the various online career services.
- To examine the advantages and disadvantages of posting and finding jobs online.
- To review the major online career services Web sites available to job seekers.
- To explore the various online services available to employers seeking to build their workforces.

What is the city but the people?
William Shakespeare

A great city is that which has the greatest men and women,
If it be a few ragged huts it is still the greatest city in the whole world.
Walt Whitman

To understand the true quality of people, you must look into their minds, and examine their pursuits and aversions.
Marcus Aurelius

The soul is made for action, and cannot rest till it be employed. Idleness is its rust. Unless it will up and think and taste and see, all is in vain.
Thomas Traherne

Outline

- **C.1 Introduction**
- **C.2 Resources for the Job Seeker**
- **C.3 Online Opportunities for Employers**
  - **C.3.1 Posting Jobs Online**
  - **C.3.2 Problems with Recruiting on the Web**
  - **C.3.3 Diversity in the Workplace**
- **C.4 Recruiting Services**
- **C.5 Career Sites**
  - **C.5.1 Comprehensive Career Sites**
  - **C.5.2 Technical Positions**
  - **C.5.3 Wireless Positions**
  - **C.5.4 Contracting Online**
  - **C.5.5 Executive Positions**
  - **C.5.6 Students and Young Professionals**
  - **C.5.7 Other Online Career Services**
- **C.6 Internet and World Wide Web Resources**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises • Works Cited

C.1 Introduction

There are approximately 40,000 career-advancement services on the Internet today.[1] These services include large, comprehensive job sites, such as **`Monster.com`** (see the upcoming **`Monster.com`** feature), as well as interest-specific job sites such as **`JustJavaJobs.com`**. Companies can reduce the amount of time spent searching for qualified employees by building recruiting features on their Web sites or establishing accounts with career sites. This results in a larger pool of qualified applicants, as online services can automatically select and reject resumes based on user-designated criteria. Online interviews, testing services and other resources also expedite the recruiting process.

Applying for a position online is a relatively new method of exploring career opportunities. Online recruiting services streamline the process and allow job seekers to concentrate their energies in careers that are of interest to them. Job seekers can explore opportunities according to geographic location, position, salary or benefits packages.

Job seekers can learn how to write resumes and cover letters, post them online and search through job listings to find the jobs that best suit their needs. *Entry-level positions*, or positions commonly sought by individuals who are entering a specific field or the job market for the first time; contracting positions; executive-level positions and middle-management-level positions are all available on the Web.

Job seekers will find a number of time-saving features when searching for jobs online. These include storing and distributing resumes digitally, e-mail notification of possible positions, salary and relocation calculators, job coaches, self-assessment tools and information on continuing education.

In this chapter, we explore online career services from the employer and employee's perspective. We suggest sites on which applications can be submitted, jobs can be searched and applicants can be reviewed. We also review services that build recruiting pages directly into e-businesses.

C.2 Resources for the Job Seeker

Finding a job online can greatly reduce the amount of time spent applying for a position. Instead of searching through newspapers and mailing resumes, job seekers can request a specific positions in specific industries through search engines. Some sites allow job seekers to setup intelligent agents to find jobs that meet their requirements. Intelligent agents are programs that search and arrange large amounts of data and report answers based on that data. When the agent finds a potential match, it sends it to the job seeker's inbox. Resumes can be stored digitally, customized quickly to meet job requirements and e-mailed instantaneously. A potential candidate also can learn more about a company by visiting its Web site. Most employment sites are free to job seekers. These sites typically generate their revenues by charging employers for posting job opportunities and by selling advertising space on their Web pages (see the **`Monster.com`** feature).

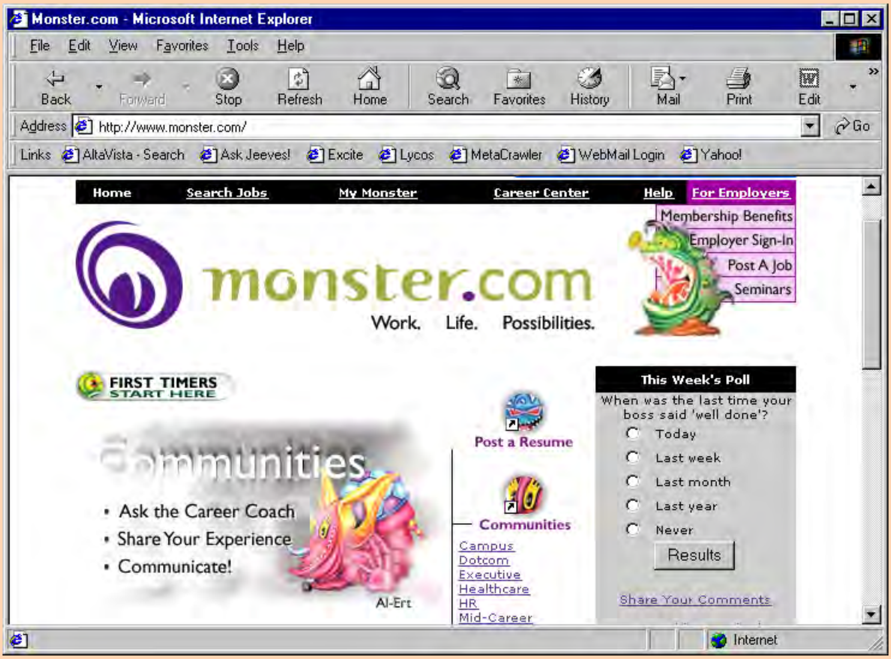
Career services, such as **`FlipDog.com`**, search a list of employer job sites to find positions. By searching links to employer Web sites, **`FlipDog.com`** is able to identify positions from companies of all sizes. This feature enables job seekers to find jobs that employers may not have posted outside the corporation's Web site.

`Monster.com`

Super Bowl ads and effective marketing have made **`Monster.com`** one of the most recognizable online brands (see Fig. C.1). In fact, in the 24 hours following Super Bowl XXXIV, 5 million job searches occurred on **`Monster.com`**.[2] The site allows people looking for jobs to post their resumes, search job listings, read advice and information about the job-search process and take proactive steps to improve their careers. These services are free to job seekers. Employers can post job listings, search resume databases and become featured employers.

Posting a resume at **`Monster.com`** is simple and free. **`Monster.com`** has a resume builder that allows users to post a resume to its site in 15–30 minutes. Each user can store up to 5 resumes and cover letters on the **`Monster.com`** server. Some companies offer their employment applications directly through the **`Monster.com`** site. **`Monster.com`** has job postings in every state and all major categories. Users can limit access to their personal identification information. As one of the leading recruiting sites on the Web, **`Monster.com`** is a good place to begin a job search or to find out more about the search process.

`Monster.com` (Cont.)


Monster.com - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address http://www.monster.com/
Go
Links AltaVista - Search Ask Jeeves! Excite Lycos MetaCrawler WebMail Login Yahoo!
Home Search Jobs My Monster Career Center Help For Employers
Membership Benefits
Employer Sign-In
Post A Job
Seminars
monster.com
Work. Life. Possibilities.
FIRST TIMERS START HERE
Communities
Ask the Career Coach
Share Your Experience
Communicate!
Al-Ert
Post a Resume
Communities
Campus
Dotcom
Executive
Healthcare
HR
Mid-Career
This Week's Poll
Results
Share Your Comments
Internet

When was the last time your boss said 'well done'?

○ Today
○ Last week
○ Last month
○ Last year
○ Never

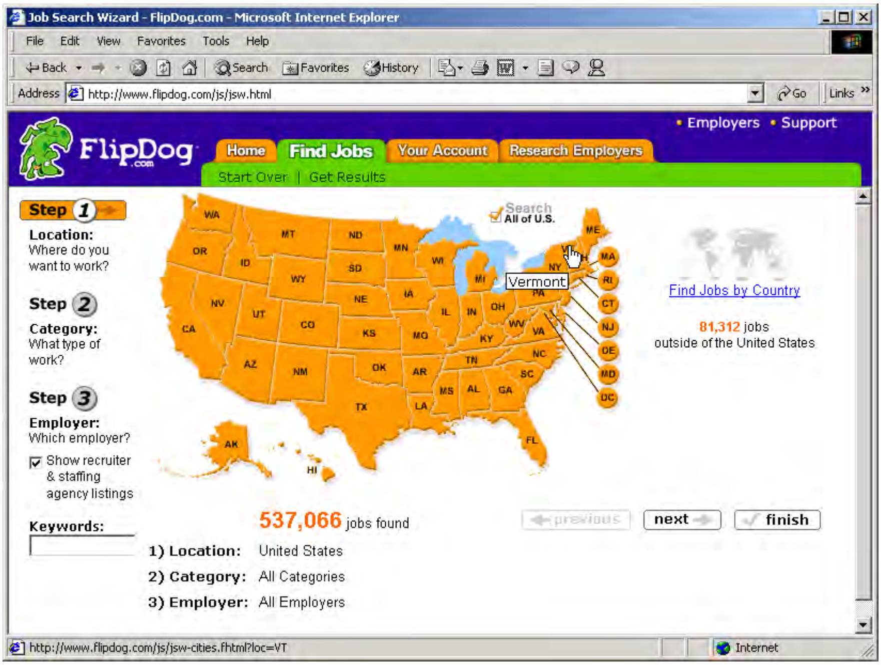
Fig. C.1 `Monster.com` home page. (Courtesy of `Monster.com`.]

Job seekers can visit **`FlipDog.com`** and choose, by state, the area in which they are looking for positions. Applicants also can conduct worldwide searches. After a user selects a region, **`FlipDog.com`** requests the user to choose a job category containing several specific positions. The user's choice causes a list of local employers to appear. The user can specify an employer or request that **`FlipDog.com`** search the employment databases for jobs offered by all employers (see Fig. C.2).

Other services, such as employment networks, also help job seekers in their search. Sites such as **`Vault.com`** (see the **`Vault.com`** feature) and **`WetFeet.com`** allow job seekers to post questions in designated chat rooms or on electronic bulletin boards about employers and positions.

C.3 Online Opportunities for Employers

Recruiting on the Internet provides several benefits over traditional recruiting. For example, Web recruiting reaches a much larger audience than posting an advertisement in a local newspaper. Given the breadth of the services provided by most online career services Web sites, the cost of posting online can be considerably less than posting positions through traditional means. Even newspapers, which depend greatly on career opportunity advertising, are starting online career sites.[3]


Job Search Wizard - FlipDog.com - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Go Links
Employers Support
FlipDog.com
Home
Find Jobs
Your Account
Research Employers
Start Over | Get Results
Search All of U.S.
WA OR ID MT ND SD WY NV CA UT CO NE KS AZ NM OK TX MN IA MO AR LA WI IL MI IN OH KY TN MS AL GA FL SC NC VA WV PA NY ME AK HI
Vermont
MA RI CT NJ DE MD DC
Find Jobs by Country
previous
next
finish
Internet

Address http://www.flipdog.com/js/jsw.html

Step 1
Location: Where do you want to work?

Step 2
Category: What type of work?

Step 3
Employer: Which employer?

☑ Show recruiter & staffing agency listings

Keywords:

81,312 jobs outside of the United States

537,066 jobs found

1) **Location:** United States
2) **Category:** All Categories
3) **Employer:** All Employers

http://www.flipdog.com/js/jsw-cities.fhtml?loc=VT

Fig. C.2 `FlipDog.com` job search. (Courtesy of `Flipdog.com`.)

Vault.com: Finding the Right Job on the Web[4]

`Vault.com` allows potential employees to seek out additional, third-party information for over 3000 companies. By visiting the *Insider Research* page, Web users have access to a profile on the company of their choice, as long as it exists in `Vault.com`'s database. In addition to `Vault.com`'s profile, there is a link to additional commentary by company employees. Most often anonymous, these messages can provide prospective employees with potentially valuable decision-making information. However, users must consider the integrity of the source. For example, a disgruntled employee may leave a posting that is not an accurate representation of the corporate culture of his or her company.

The `Vault.com` *Electronic Watercooler*™ is a message board that allows visitors to post stories, questions and concerns and to advise employees and job seekers. In addition, the site provides e-newsletters and feature stories designed to help job seekers in their search. Individuals seeking information on business, law and graduate schools can also find information on `Vault.com`.

Job-posting and career-advancement services for the job seeker are featured on `Vault.com`. These services include *VaultMatch*, a career service that e-mails job postings as requested, and *Salary Wizard*™, which helps job seekers determine the salary they are worth. Online guides with advice for fulfilling career ambitions are also available.

Vault.com: Finding the Right Job on the Web[4] (Cont.)

Employers can also use the site. *HR Vault*, a feature of `Vault.com`, provides employers with a free job-posting site. It offers career-management advice, employer-to-employee relationship management and recruiting resources.

e-Fact C.1

According to Forrester Research, 33 percent of today's average company's hiring budget goes toward online career services, while the remaining 66 percent is used for traditional recruiting mechanisms. Online use is expected to increase to 42 percent by 2004, while traditional mechanisms may be reduced to 10 percent.[5]

Generally, jobs posted online are viewed by a larger number of job seekers than jobs posted through traditional means. However, it is important not to overlook the benefits of combining online efforts with human-to-human interaction. There are many job seekers who are not yet comfortable with the process of finding a job online. Often, online recruiting is used as a means of freeing up a recruiter's time for the interviewing process and final selection.

e-Fact C.2

Cisco Systems cites a 39 percent reduction in cost-per-hire expenses, and a 60 percent reduction in the time spent hiring.[6]

C.3.1 Posting Jobs Online

When searching for job candidates online, there are many things employers need to consider. The Internet is a valuable tool for recruiting, but one that takes careful planning to acquire the best results. It provides a good supplementary tool, but should not be considered the complete solution for filling positions. Web sites, such as WebHire (`www.webhire.com`), enhance a company's online employment search (see the WebHire feature).

There are a variety of sites that allow employers to post jobs online. Some of these sites require a fee, which generally runs between $100–$200. Postings typically remain on the Web site for 30–60 days. Employers should be careful to post to sites that are most likely to be visited by eligible candidates. As we discovered in the previous section, there are a variety of online career services focused on specific industries, and many of the larger, more comprehensive sites have categorized their databases by job category.

When designing a posting, the recruiter should consider the vast number of postings already on the Web. Defining what makes the job position unique, including information such as benefits and salary, might convince a qualified candidate to further investigate the position (see Fig. C.3).[7]

`HotJobs.com` career postings are cross-listed on a variety of other sites, thus increasing the number of potential employees who see the job listings. Like `Monster.com` and `jobfind.com`, `HotJobs.com` requires a fee per listing. Employers also have the option of becoming `HotJobs.com` members. Employers can gain access to HotJob's *Private Label Job Boards* (private corporate employment sites), online recruiting technology and online career fairs.

WebHire™[8]

Designed specifically for recruiters and employers, WebHire is a multifaceted service that provides employers with *end-to-end recruiting solutions*. The service offers job-posting services as well as candidate searches. The most comprehensive of the services, *WebHire™ Enterprise*, locates and ranks candidates found through resume-scanning mechanisms. Clients will also receive a report indicating the best resources for their search. Other services available through the *WebHire™ Employment Services Network* include preemployment screening, tools for assessing employees' skill levels and information on compensation packages. An employment law advisor helps organizations design interview questions.

WebHire™ Agent is an intelligent agent that searches for qualified applicants based on job specifications. When WebHire Agent identifies a potential candidate, an e-mail is sent to the candidate to generate interest. WebHire Agent then ranks applicants according to the skills information it gains from the Web search; the information is stored so that new applicants are distinguished from those who have already received an e-mail from the site.

Yahoo!® Resumes, a feature of WebHire, allows recruiters to find potential employees by typing in keywords on the Yahoo! Resumes search engine. Employers can purchase a year's membership to the recruiting solution for a flat fee; there are no per-use charges.

Job Seeker's Criteria

Position (responsibilities)
Salary
Location
Benefits (health, dental, stock options)
Advancement
Time Commitment
Training Opportunities
Tuition Reimbursement
Corporate Culture

Fig. C.3 List of a job seeker's criteria.

Boston Herald *Job Find* (**`www.jobfind.com`**) also charges employers to post on its site. The initial fee entitles the employer to post up to three listings. Employers have no limitations on the length of their postings.

Other Web sites providing employers with employee recruitment services include **`CareerPath.com`**, America's Job Bank (**`www.ajb.dni.us/employer`**), CareerWeb (**`www.cweb.com`**), **`Jobs.com`** and **`Career.com`**.

C.3.2 Problems with Recruiting on the Web

The large number of applicants presents a challenge to both job seekers and employers. On many recruitment sites, matching resumes to positions is conducted by *resume-filtering software*. The software scans a pool of resumes for keywords that match the job description. While this software increases the number of resumes that receive attention, it is not a foolproof system. For example, the resume-filtering software might overlook someone with similar skills to those listed in the job description, or someone whose abilities would enable them to learn the skills required for the position. Digital transmissions can also create problems because certain software platforms are not always acceptable by the recruiting software. This sometimes results in an unformatted transmission, or a failed transmission.

A lack of confidentiality is another disadvantage of online career services. In many cases, a job candidate will want to search for job opportunities anonymously. This reduces the possibility of offending the candidate's current employer. Posting a resume on the Web increases the likelihood that the candidate's employer might come across it when recruiting new employees. The traditional method of mailing resumes and cover letters to potential employers does not impose the same risk.

According to recent studies, the number of individuals researching employment positions through traditional means, such as referrals, newspapers and temporary agencies, far outweighs the number of job seekers researching positions through the Internet.[9] Optimists feel, however, that this disparity is largely due to the early stages of e-business development. Given time, online career services will become more refined in their posting and searching capabilities, decreasing the amount of time it takes for a job seeker to find jobs and employers to fill positions.

C.3.3 Diversity in the Workplace

Every workplace inevitably develops its own culture. Responsibilities, schedules, deadlines and projects all contribute to a working environment. Perhaps the most defining elements of a *corporate culture* are the employees. For example, if all employees were to have the same skills, same backgrounds and the same ideas, the workplace would lack diversity. It also might lack creativity and enthusiasm. One way to increase the dynamics of an organization is to employ people of different backgrounds and cultures.

The Internet hosts demographic-specific sites for employers seeking to increase diversity in the workplace. By recruiting people from different backgrounds, new ideas and perspectives are brought forth, helping businesses meet the needs of a larger, more diverse target audience.[10]

`Blackvoices.com` and **`hirediversity.com`** are demographic-specific Web sites. BlackVoices™, which functions primarily as a portal (a site offering news, sports and weather information, as well as Web searches), features job searching capabilities and the ability for prospective employees to post resumes. HireDiversity is divided into several categories, including opportunities for African Americans, Hispanics and women. Other online recruiting services place banner advertisements on ethnic Web sites for companies seeking diverse workforces.

The Diversity Directory (**`www.mindexchange.com`**) offers international career-searching capabilities. Users selecting the **Diversity** site can find job opportunities, information and additional resources to help them in their career search. The site can be searched according to demographics (African American, Hispanic, alternative lifestyle, etc.) or by subject (employer, position, etc.) via hundreds of links. Featured sites include **`BilingualJobs.com`**, *Latin World* and *American Society for Female Entrepreneurs*.

Many sites have sections dedicated to job seekers with disabilities. In addition to providing job-searching capabilities, these sites include additional resources, such as equal opportunity documents and message boards. The *National Business and Disability Council* (*NBDC*) provides employers with integration and accessibility information for employing people with disabilities, and the site also lists opportunities for job seekers.

C.4 Recruiting Services

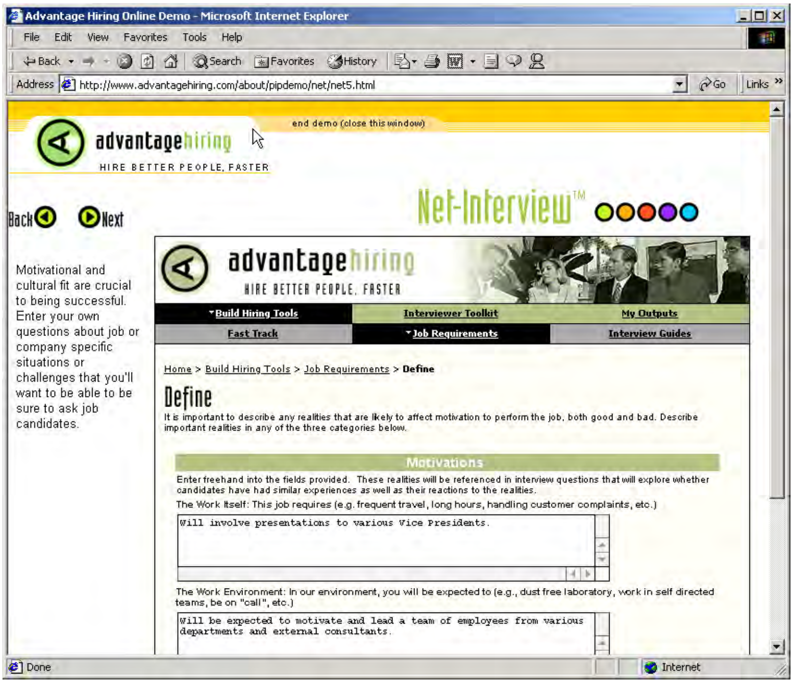
There are many services on the Internet that help employers match individuals to positions. The time saved by conducting preliminary searches on the Internet can be dedicated to interviewing qualified candidates and making the best matches possible.

Advantage Hiring, Inc. (**`www.advantagehiring.com`**) provides employers with a resume-screening service. When a prospective employee submits a resume for a particular position, Advantage Hiring, Inc. presents *Net-Interview*™, a small questionnaire to supplement the information presented on the resume. The site also offers *SiteBuilder*, a service that helps employers build an employee recruitment site. An online demonstration can be found at **`www.advantagehiring.com`**. The demonstration walks the user through the Net-Interview software, as well as a number of other services offered by Advantage Hiring (see Fig. C.4).

`Recruitsoft.com` is an application service provider (ASP) that offers companies recruiting software on a *pay-per-hire* basis (Recruitsoft receives a commission on hires made via its service). *Recruiter WebTop*™ is the company's online recruiting software. It includes features such as Web-site hosting, an employee-referral program, skill-based resume screening, applicant-tracking capabilities and job-board posting capabilities. A demonstration of Recruiter WebTop's *Corporate Recruiting Solutions* can be found at **`www.recruitsoft.com/process`**. Other online recruiting services include **`Hire.com`**, and **`Futurestep.com`**™.

The Internet also provides employers with a cost-effective means of testing their prospective employees in such categories as decision making, problem solving and personality. Services such *eTest* help to reduce the cost of in-house testing and to make the interview process more effective. Test results, given in paragraph form, present employers with the interested individual's strengths and weaknesses. Based on these results, the report suggests interview methods, such as asking *open-ended questions*, which are questions that require more than a "yes" or "no" response. Sample reports and a free-trial test can be found at **`www.etest.net`**.

Employers and job seekers can also find career placement exercises at **`www.advisorteam.net/User/ktsintro.asp`**. Some of these services require a fee. The tests ask several questions regarding the individual's interests and working style. Results help candidates determine the best career for their skills and interests.


Advantage Hiring Online Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.advantagehiring.com/about/pipdemo/net/net5.html
Go Links
end demo (close this window)
advantagehiring
HIRE BETTER PEOPLE, FASTER
Net-Interview™
Back Next
Build Hiring Tools
Interviewer Toolkit
My Outputs
Fast Track
Job Requirements
Interview Guides
Home > Build Hiring Tools > Job Requirements > Define
Done
Internet

Motivational and cultural fit are crucial to being successful. Enter your own questions about job or company specific situations or challenges that you'll want to be able to be sure to ask job candidates.

Define

It is important to describe any realities that are likely to affect motivation to perform the job, both good and bad. Describe important realities in any of the three categories below.

Motivations

Enter freehand into the fields provided. These realities will be referenced in interview questions that will explore whether candidates have had similar experiences as well as their reactions to the realities.

The Work Itself: This job requires (e.g. frequent travel, long hours, handling customer complaints, etc.)

Will involve presentations to various Vice Presidents.

The Work Environment: In our environment, you will be expected to (e.g., dust free laboratory, work in self directed teams, be on "call", etc.)

Will be expected to motivate and lead a team of employees from various departments and external consultants.

Fig. C.4 Advantage Hiring, Inc.'s Net-Interview™ service. (Courtesy of Advantage Hiring, Inc.)

C.5 Career Sites

Online career sites can be comprehensive or industry specific. In this section, we explore a variety of sites on the Web that accommodate the needs of both the job seeker and the employer. We review sites offering technical positions, free-lancing opportunities and contracting positions.

C.5.1 Comprehensive Career Sites

As mentioned previously, there are many sites on the Web that provide job seekers with career opportunities in multiple fields. `Monster.com` is the largest of these sites, attracting the greatest number of unique visitors per month. Other popular online recruiting sites include `JobsOnline.com`, `HotJobs.com`, `www.jobtrak.com` (a `Monster.com` site) and `Headhunter.net`.

Searching for a job online can be a conducted in a few steps. For example, during an initial visit to `JobsOnline.com`, a user is required to fill out a registration form. The form requests basic information, such as name, address and area of interest. After registering, members can search through job postings according to such criteria as job category, location and the number of days the job has been posted. Contact information is provided for additional communication.

C.5.2 Technical Positions

Technical positions are becoming widely available as the Internet grows more pervasive. Limited job loyalty and high turnover rates in technical positions allow job seekers to find jobs that best suit their needs and skills. Employers are required to rehire continuously to keep positions filled and productivity levels high. The amount of time for an employer to fill a technical position can be greatly reduced by using an industry-specific site. Career sites designed for individuals seeking technical positions are among the most popular online career sites. In this section, we review several sites that offer recruiting and hiring opportunities for technical positions.

e-Fact C.3

It costs a company 25 percent more to hire a new technical employee than it does to pay an already employed individual's salary.[11]

Dice.com (**www.dice.com**) is a recruiting Web site that focuses on technical fields. Company fees are based on the number of jobs the company posts and the frequency with which the postings are updated. Job seekers can post their resumes and search the job database for free. **JustTechJobs.com** directs job seekers toward 39 specific computer technologies for their job search. Language-specific sites include **JustJavaJobs.com**, **JustCJobs.com** and **JustPerlJobs.com**. Hardware, software and communications technology sites are also available. Other technology recruiting sites include **Hire-Ability.com**, and **HotDispatch.com**.

C.5.3 Wireless Positions

The wireless industry is developing rapidly. According to **WirelessResumes.com**, the number of wireless professionals is 328,000. This number is expected to increase 40 percent each year for the next five years. To accommodate this growth, and the parallel demand for professionals, **WirelessResumes.com** has created an online career site specifically for the purpose of filling wireless jobs (see the **WirelessResumes.com** feature).

WirelessResumes.com: *Filling Wireless Positions*

WirelessResumes.com is an online career site focused specifically on matching wireless professionals with careers in the industry. This narrow focus enables businesses to locate new employees quickly—reducing the time and expense attached to traditional recruiting methods. Similarly, candidates can limit their searches to precisely the job category of interest. Wireless carriers, device manufacturers, WAP and Bluetooth developers, e-commerce companies and application service providers (ASPs) are among those represented on the site.

In addition to searching for jobs and posting a resume, **WirelessResumes.com** provides job seekers with resume writing tips, interviewing techniques, relocation tools and assistance in obtaining a Visa or the completion of other necessary paperwork. Employers can use the site to search candidates and post job opportunities.

The Caradyne Group (`www.pcsjobs.com`), an executive search firm, connects job seekers to employers in the wireless technology field. Interested job seekers must first fill out a “Profile Questionnaire.” This information is then entered into The Caradyne Group’s database and is automatically matched to an open position in the job seeker’s field of expertise. If there are no open positions, a qualified consultant from The Caradyne Group will contact the job seeker for further a interview and discussion.

C.5.4 Contracting Online

The Internet also serves as a forum for job seekers to find employment on a project-by-project basis. *Online contracting services* allow businesses to post positions for which they wish to hire outside resources, and individuals can identify projects that best suit their interests, schedules and skills.

e-Fact C.4

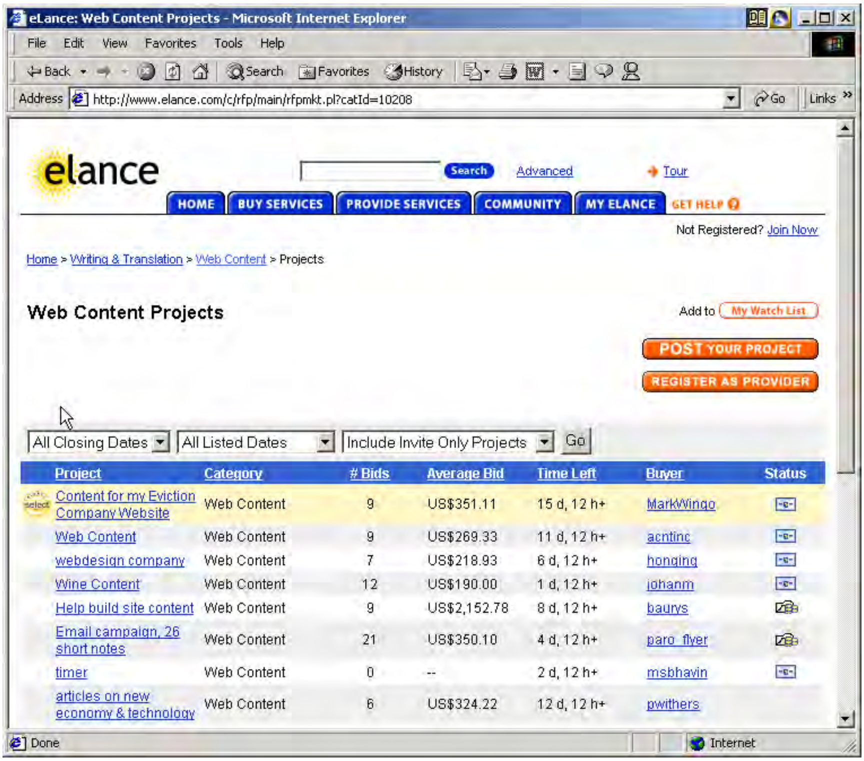
Approximately six percent of America’s workforce falls into the category of independent contractor.[12]

`Guru.com` (`www.guru.com`) is a recruiting site for contract employees. Independent contractors, private consultants and trainers use `guru.com` to find short-term and long-term contract assignments. Tips, articles and advice are available for contractors who wish to learn more about their industry. Other sections of the site teach users how to manage their businesses, buy the best equipment and deal with legal issues. `Guru.com` includes an online store where contractors can buy products associated with small-business management, such as printing services and office supplies. Companies wishing to hire contractors must register with `guru.com`, but individuals seeking contract assignments do not.

`Monster.com`’s Talent Market™ offers online auction-style career services to free agents. Interested users design a profile, listing their qualifications. After establishing a profile, free agents “Go Live” to start the bidding on their services. The bidding lasts for five days during which users can view the incoming bids. At the close of five days, the user can choose the job of his or her choice. The service is free for users, and bidding employers pay a commission on completed transactions.

`eLance.com` is another site where individuals can find contracting work. Interested applicants can search eLance’s database by category, including business, finance and marketing (Fig. C.5). These projects, or *requests for proposals* (RFPs), are posted by companies worldwide. When users find projects for which they feel qualified, they submit bids on the projects. Bids must contain a user’s required payment, a statement detailing the user’s skills and a feedback rating drawn from other projects on which the user has worked. If a user’s bid is accepted, the user is given the project, and the work is conducted over eLance’s file-sharing system, enabling both the contractor and the employer to contact one another quickly and easily. For an online demonstration, visit `www.elance.com` and click on the **take a tour...** link.

Other Web sites that provide contractors with projects and information include eWork® Exchange (`www.ework.com`), `MBAFreeAgent.com`, `Aquent.com` and `WorkingSolo.com`.


eLance: Web Content Projects - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.elance.com/c/rfp/main/rfpmkt.pl?catId=10208
Go Links
elance
Search
Advanced
Tour
HOME
BUY SERVICES
PROVIDE SERVICES
COMMUNITY
MY ELANCE
GET HELP
Not Registered? Join Now
Add to My Watch List
POST YOUR PROJECT
REGISTER AS PROVIDER
Go
Done
Internet

Home > Writing & Translation > Web Content > Projects

Web Content Projects

All Closing Dates | All Listed Dates | Include Invite Only Projects

Project	Category	# Bids	Average Bid	Time Left	Buyer	Status
Content for my Eviction Company Website	Web Content	9	US$351.11	15 d, 12 h+	MarkWingo	
Web Content	Web Content	9	US$269.33	11 d, 12 h+	acntinc	
webdesign company	Web Content	7	US$218.93	6 d, 12 h+	honging	
Wine Content	Web Content	12	US$190.00	1 d, 12 h+	johanm	
Help build site content	Web Content	9	US$2,152.78	8 d, 12 h+	baurys	
Email campaign, 26 short notes	Web Content	21	US$350.10	4 d, 12 h+	paro_flyer	
timer	Web Content	0	--	2 d, 12 h+	msbhavin	
articles on new economy & technology	Web Content	6	US$324.22	12 d, 12 h+	pwithers	

Fig. C.5 `eLance.com` request for proposal (RFP) example. (Courtesy of eLance, Inc.]

C.5.5 Executive Positions

In this section, we discuss the advantages and disadvantages of finding an executive position online. Executive career advancement sites usually include many of the features found on comprehensive job-search sites. Searching for an executive position online differs from finding an entry-level position online. The Internet allows individuals to continually survey the job market. However, candidates for executive-level positions must exercise a higher level of caution when determining who is able to view their resume. Applying for an executive position online is an extensive process. As a result of the high level of scrutiny passed on a candidate during the hiring process, the initial criteria presented by an executive level candidate often are more specific than the criteria presented by the first-time job seeker. Executive positions often are difficult to fill, due to the high demands and large amount of experience required for the jobs.

SixFigureJobs (**`www.sixfigurejobs.com`**) is a recruitment site designed for experienced executives. Resume posting and job searching is free to job seekers. Other sites, including **`www.execunet.com`**, **`Monster.com`**'s ChiefMonster™ (**`www.chiefmonster.com`**) and **`www.nationjob.com`** are designed for helping executives find positions.

C.5.6 Students and Young Professionals

The Internet provides students and young professionals with tools to get them started in the job market. Individuals still in school and seeking internships, individuals who are just graduating and individuals who have been in the workforce for a few years make up the target market. Additional tools specifically designed for this *demographic* (a population defined by a specific characteristic) are available. For example, journals kept by previous interns provide prospective interns with information regarding what to look for in an internship, what to expect and what to avoid. Many sites will provide information to lead young professionals in the right direction, such as matching positions to their college or university major.

`Experience.com` is a career services Web site geared toward the younger population. Members can search for positions according to specific criteria, such as geographic location, job category, keywords, commitment (i.e. full time, part time, internship), amount of vacation and amount of travel time. After applicants register, they can send their resumes directly to the companies posted on the site. In addition to the resume, candidates provide a personal statement, a list of applicable skills and their language proficiency. Registered members also receive access to the site's *Job Agent*. Up to three Job Agents can be used by each member. The agents search for available positions, based on the criteria posted by the member. If a match is made, the site contacts the candidate via e-mail.[13,14]

`Internships.wetfeet.com` helps students find internships. In addition to posting a resume and searching for an internship, students can use the relocation calculator to compare the cost of living in different regions. Tips on building resumes and writing essays are provided. The *City Intern* program provides travel, housing and entertainment guides to interns interviewing or accepting a position in an unfamiliar city, making them feel more at home in a new location.

In addition to its internship locators, undergraduate, graduate, law school, medical school and business school services, the Princeton Review's Web site (**`www.review.com`**) offers career services to graduating students. While searching for a job, students and young professionals can also read through the site's news reports or even increase their vocabulary by visiting the "word for the day." Other career sites geared toward the younger population include **`campuscareercenter.com`**, **`brassring-campus.com`** and **`collegegrad.com`**.

C.5.7 Other Online Career Services

In addition to Web sites that help users find and post jobs online, there are a number of Web sites that offer features that will enhance searches, prepare users to search online, help applicants design resumes or help users calculate the cost of relocating.

`Salary.com` helps job seekers gauge their expected income, based on position, level of responsibility and years of experience. The search requires job category, ZIP code and specific job title. Based on this information, the site will return an estimated salary for an individual living in the specified area and employed in the position described. Estimates are returned based on the average level of income for the position.

In addition to helping applicants find employment, **`www.careerpower.com`** provides individuals with tests that will help them realize their strengths, weaknesses, values, skills and personality traits. Based on the results, which can be up to 10–12 pages per test, users can best decide what job categories they are qualified for and what career choice will be best suited to their personal ambitions. The service is available for a fee.

InterviewSmart™ is another service offered through CareerPower that prepares job seekers of all levels for the interviewing process. The service can be downloaded for a minimal fee or can be used on the Web for free. Both versions are available at **`www.careerpower.com/CareerPerfect/interviewing.htm#is.start.anchor`**.

Additional services will help applicants find positions that meet their unique needs, or design their resumes to attract the attention of specific employers. **`Dogfriendly.com`**, organized by geographic location, helps job seekers find opportunities that allow them to bring their pets to work, and **`cooljobs.com`** is a searchable database of unique job opportunities.

C.6 Internet and World Wide Web Resources

Information Technology (IT) Career Sites

`www.dice.com`
This is a recruiting Web site that focuses on the computer industry.

`www.guru.com`
This is a recruiting site for contract employees. Independent contractors, private consultants and trainers can use **`guru.com`** to find short-term and long-term work.

`www.hallkinion.com`
This is a Web recruiting service for individuals seeking IT positions.

`www.techrepublic.com`
This site provides employers and job seekers with recruiting capabilities and information regarding developing technology.

`www.justcomputerjobs.com`
This site serves as a portal with access to language-specific sites, including Java, Perl, C and C++.

`www.hotdispatch.com`
This forum provides software developers with the opportunity to share projects, discuss code and ask questions.

`www.techjobs.bizhosting.com/jobs.htm`
This site directs job seekers to links of numerous technological careers listed by location, internet, type of field, etc.

Career Sites

`www.careerbuilder.com`
A network of career sites, including IT Careers, *USA Today* and MSN, CareerBuilder attracts 3 million unique job seekers per month. The site provides resume-builder and job-searching agents.

`www.recruitek.com`
This free site caters to jobs seekers, employers and contractors.

`www.monster.com`
This site, the largest of the online career sites, allows people looking for jobs to post their resumes, search job listings and read advice and information about the job-search process. It also provides a variety of recruitment services for employers.

`www.jobsonline.com`
Similar to **`Monster.com`**, this site provides opportunities for job seekers and employers.

`www.hotjobs.com`
This online recruiting site offers cross-listing possibilities on additional sites.

`www.jobfind.com`
This job site is an example of locally targeted job-search resources. **`JobFind.com`** targets the Boston area.

`www.flipdog.com`
This site allows online job candidates to search for career opportunities. It employs intelligent agents to scour the Web and return jobs matching the candidate's request.

`www.cooljobs.com`
This site highlights unique job opportunities.

`www.inetsupermall.com`
This site aids job searchers in creating professional resumes and connecting with employers.

`www.wirelessnetworksonline.com`
This site helps connect job searchers to careers for which they are qualified.

`www.careerweb.com`
This site highlights featured employers and jobs and allows job seekers and employers to post and view resumes, respectively.

`www.jobsleuth.com`
On this site job seekers can fill out a form that indicates their desired field of employment. Job Sleuth™ searches the Internet and returns potential matches to the user's inbox. The service is free.

`www.ajb.org`
America's Job Bank is an online recruiting service provided through the Department of Labor and the state employment service. Searching for and posting positions on the site are free.

Executive Positions

`www.sixfigurejobs.com`
This is a recruitment site designed for experienced executives.

`www.leadersonline.com`
This career services Web site offers confidential job searches for mid-level professionals. Potential job matches are e-mailed to job candidates.

`www.ecruitinginc.com`
This site is designed to search for employees for executive positions.

Diversity

`www.latpro.com`
This site is designed for Spanish-speaking and Portuguese-speaking job seekers. In addition to providing resume-posting services, the site enables job seekers to receive matching positions via e-mail. Advice and information services are available.

`www.blackvoices.com`
This portal site hosts a career center designed to match African American job seekers with job opportunities.

`www.hirediversity.com`
In addition to services for searching for and posting positions, resume-building and updating services are also available on this site. The site targets a variety of demographics including African Americans, Asian Americans, people with disabilities, women and Latin Americans.

People with Disabilities

`www.halftheplanet.com`
This site represents people with disabilities. The site is large and includes many different resources and information services. A special section is dedicated to job seekers and employers.

`www.wemedia.com`
This site is designed to meet the needs of people with disabilities. It includes a section for job seekers and employers.

`www.disabilities.com`
This site provides users with a host of links to information resources on career opportunities.

`www.mindexchange.com`
The diversity section of this site provides users with several links to additional resources regarding people with disabilities and employment.

`www.usdoj.gov/crt/ada/adahom1.htm`
This is the Americans with Disabilities Act home page.

`www.abanet.org/publicserv/mental.html`
This is the Web site for The Commission on Mental and Physical Disability Law.

`janweb.icdi.wvu.edu`
The Job Accommodation Web site offers consulting services to employers regarding integration of people with disabilities into the workplace.

General Resources

`www.vault.com`
This site provides potential employees with “insider information” on over 3000 companies. In addition, job seekers can search through available positions and post and answer questions on the message board.

`www.wetfeet.com`
Similar to **`vault.com`**, this site allows visitors to ask questions and receive “insider information” on companies that are hiring.

Special Interest

`www.eharvest.com/careers/`
This Web site provides job seekers interested in agricultural positions with online career services.

`www.opportunitynocs.org`
This career services site is for both employers and job seekers interested in non-profit opportunities.

`www.experience.com`
This Web site is designed specifically for young professionals and students seeking full-time, part-time and internship positions.

`www.internships.wetfeet.com`
Students seeking internships can search job listings on this site. It also features City Intern, to help interns become acquainted with a new location.

`www.brassringcampus.com`
This site provides college grads and young professionals with less than five years of experience with job opportunities. Additional features help users buy cars or find apartments.

Online Contracting

`www.ework.com`
This online recruiting site matches outside contractors with companies needing project specialists. Other services provided through eWork include links to online training sites, benefits packages and payment services and online meeting and management resources.

`www.elance.com`
Similar to **`eWork.com`**, eLance matches outside contractors with projects.

`www.MBAFreeAgent.com`
This site is designed to match MBAs with contracting opportunities.

`www.aquent.com`
This site provides access to technical contracting positions.

`www.WorkingSolo.com`
This site helps contractors begin their own projects.

Recruiting Services

`www.advantagehiring.com`
This site helps employers screen resumes.

`www.etest.net`
This site provides employers with testing services to assess the strengths and weaknesses of prospective employees. This information can be used for better hiring strategies.

`www.hire.com`
`Hire.com`'s eRecruiter is an application service provider that helps organizations streamline their Web-recruiting process.

`www.futurestep.com`
Executives can register confidentially at **`Futurestep.com`** to be considered for senior executive positions. The site connects registered individuals to positions. It also offers career management services.

`www.webhire.com`
This site provides employers with end-to-end recruiting solutions.

Wireless Career Resources

`www.wirelessresumes.com/`
This site connects employers and job seekers with resumes that focus on jobs revolving around wireless technology.

`www.msua.org/job.htm`
This site contains links to numerous wireless job-seeking Web sites.

`www.wiwc.org`
This site's focus is wireless communication job searching for women.

`www.firstsearch.com`
At this site a job seeker is able to discover part-time, full-time and salary-based opportunities in the wireless industry.

`www.pcsjobs.com`
This is the site for The Caradyne Group, which is an executive search firm that focuses on finding job seekers wireless job positions.

`www.cnijoblink.com`
CNI Career Networks offers confidential, no-charge job placement in the wireless and telecommunications industries.

SUMMARY

- The Internet can improve an employer's ability to recruit employees and help users find career opportunities worldwide.
- Job seekers can learn how to write a resume and cover letter, post them online and search through job listings to find the jobs that best suit their needs.
- Employers can post jobs that can be searched by an enormous pool of applicants.
- Job seekers can store and distribute resumes digitally, receive e-mail notification of possible positions, use salary and relocation calculators, consult job coaches and use self-assessment tools when searching for a job on the Web.
- There are approximately 40,000 career-advancement services on the Internet today.
- Finding a job online can greatly reduce the amount of time spent applying for a position. Potential candidates can also learn more about a company by visiting its Web site.
- Most sites are free to job seekers. These sites typically generate their revenues by charging employers who post their job opportunities, and by selling advertising space on their Web pages.
- Sites such as `Vault.com` and `WetFeet.com` allow job seekers to post questions about employers and positions in chat rooms and on bulletin boards.
- On many recruitment sites, the match of a resume to a position is conducted with resume-filtering software.
- A lack of confidentiality is a disadvantage of online career services.
- According to recent studies, the number of individuals researching employment positions through means other than the Internet, such as referrals, newspapers and temporary agencies, far outweighs the number of Internet job seekers.
- Career sites designed for individuals seeking technical positions are among the most popular online career sites.
- Online contracting services allow businesses to post positions for which they wish to hire outside resources, and allow individuals to identify projects that best suit their interests, schedules and skills.
- The Internet provides students and young professionals with some of the necessary tools to get them started in the job market. The target market is made up of individuals still in school and seeking internships, individuals who are just graduating and individuals who have been in the workforce for a few years.
- There are a number of Web sites that offer features that enhance job searches, prepare users to search online, help design applicants' resumes or help users calculate the cost of relocating.
- Web recruiting reaches a much larger audience than posting an advertisement in the local newspaper.
- There are a variety of sites that allow employers to post jobs online. Some of these sites require a fee, which generally runs between $100–$200. Postings remain on the Web site for approximately 30–60 days.
- Employers should try to post to sites that are most likely to be visited by eligible candidates.

- When designing a job posting, defining what makes a job position unique and including information such as benefits and salary might convince a qualified candidate to further investigate the position.
- The Internet hosts demographic-specific sites for employers seeking to increase diversity in the workplace.
- The Internet has provided employers with a cost-effective means of testing their prospective employees in such categories as decision making, problem solving and personality.

TERMINOLOGY

corporate culture
demographic
end-to-end recruiting solutions
entry-level position
online contracting service
open-ended question
pay-per-hire
request for proposal (RFP)
resume-filtering software

SELF-REVIEW EXERCISES

C.1 State whether each of the following is *true* or *false*, if *false*, explain why.

a) Online contracting services allow businesses to post job listings for specific projects that can be viewed by job seekers over the Web.

b) Employment networks are Web sites designed to provide information on a selected company to better inform job seekers of the corporate environment.

c) The large number of applications received over the Internet is considered an advantage by most online recruiters.

d) There is a greater number of individuals searching for work on the Web than through all other mediums combined.

e) Sixteen percent of America's workforce is categorized as independent contractors.

C.2 Fill in the blanks in each of the following statements.

a) There are approximately __________ online career services Web sites on the Internet today.

b) The Internet hosts demographic-specific sites for employers seeking to increase __________ in the workplace.

c) In the 24 hours following the Super Bowl, __________ job searches occurred on `Monster.com`.

d) Many recruitment sites use __________ to filter through received resumes.

e) Employers should try to post to sites that are most likely to be visited by __________ candidates.

ANSWERS TO SELF-REVIEW EXERCISES

C.1 a) True. b) True. c) False. The large number of applicants reduces the amount of time a recruiter can spend interviewing and making decisions. Despite screening processes, many highly qualified applicants can be overlooked. d) False. The number of individuals researching employment positions through other means, such as referrals, newspapers and temporary agencies, far outweighs the number of Internet job seekers. e) False. Six percent of America's workforce is categorized as independent consultants.

C.2 a) 40,000. b) diversity. c) 5 million. d) resume-filtering software. e) eligible.

EXERCISES

C.3 State whether each of the following is *true* or *false*, if *false*, explain why.

a) RFP is the acronym for request for proposal.
b) The Internet has provided employers with a cost-effective means of testing their prospective employees in such categories as decision making, problem solving and personality.
c) Online job recruiting can completely replace other means of hiring employees.
d) Posting a job online is less expensive than placing ads in more traditional media.
e) A lack of confidentiality is a disadvantage of online career services.

C.4 Fill in the blanks in each of the following:

a) Finding a job online can greatly __________ the amount of time spent applying for a position.
b) __________ is an example of a Web site in which contractors can bid on projects.
c) When designing a job posting, defining what makes the position unique and including information such as __________ and __________ might convince a qualified candidate to further investigate the position.
d) The Internet hosts __________ for employers seeking to increase diversity in the workplace.
e) The Internet provides employers with a cost-effective means of testing their prospective employees in such categories as __________, __________ and __________.

C.5 Define the following

a) Corporate culture.
b) Pay-per-hire.
c) Request for proposal (RFP).
d) Resume-filtering software.

C.6 (*Class discussion*). In this chapter, we discuss the short-comings and advantages of recruiting on the Internet. Using the text, additional reading material and personal accounts answer the following questions. Be prepared to discuss your answers.

a) Do you think finding a job is easier on the Web? Why or why not?
b) What disadvantages can you identify?
c) What are some of the advantages?
d) Which online recruiting services do you think will be most successful? Why?

C.7 Many of the career services Web sites we have discussed in this chapter offer resume-building capabilities. Begin building your resume, choosing an objective that is of interest to you. Think of your primary concerns. Are you searching for a paid internship or a volunteer opportunity? Do you have a specific location in mind? Do you have an opportunity for future employment? Are stock options important to you? Find several entry-level jobs that meet your requirements. Write a short summary of your results. Include any obstacles and opportunities.

C.8 In this chapter, we have discussed online contracting opportunities. Visit eLance (**`www.elance.com`**) and search the requests for proposals for contracting opportunities that interest you or visit **`guru.com`** and create a profile.

C.9 In this chapter, we have discussed many career services Web sites. Choose three sites. Explore the opportunities and resources offered by the sites. Visit any demonstrations, conduct a job search, build your resume and calculate your salary or relocation expenses. Answer the following questions.

a) Which site provides the best service? Why?
b) What did you like? Dislike?
c) Write a brief summary of your findings, including descriptions of any features that you would add.

WORKS CITED

The notation <**`www.domain-name.com`**> indicates that the citation is for information found at the Web site.

1. J. Gaskin, "Web Job Sites Face Tough Tasks," *Inter@ctive Week* 14 August 2000: 50.

2. J. Gaskin, 50.

3. M. Berger, "Jobs Supermarket," *Upside* November 2000: 224.

4. **`<www.vault.com>.`**

5. M. Berger, 224.

6. Cisco Advertisement, *The Wall Street Journal* 19 October 2000: B13.

7. M. Feffer, "Posting Jobs on the Internet," 18 August 2000 **`<www.webhire.com/hr/spotlight.asp>.`**

8. **`<www.webhire.com>.`**

9. J. Gaskin, 51.

10. C. Wilde, "Recruiters Discover Diverse Value in Web Sites," *Information Week* 7 February 2000: 144.

11. A.K. Smith, "Charting Your Own Course," *U.S. News and World Report* 6 November 2000: 58.

12. D. Lewis, "Hired! By the Highest Bidder," *The Boston Globe* 9 July 2000: G1**.**

13. **`<www.experience.com>.`**

14. M. French, "Experience Inc., E-Recruiting for Jobs for College Students," *Mass High Tec*h 7 February–13 February 2000: 29.

D

Visual Studio .NET Debugger

Objectives

- To understand syntax and logic errors.
- To become familiar with the Visual Studio .NET debugging tools.
- To understand the use of breakpoints to suspend program execution.
- To be able to examine data using expressions in the debugging windows.
- To be able to debug methods and objects.

And often times excusing of a fault
Doth make the fault the worse by the excuse.
William Shakespeare

To err is human, to forgive divine.
Alexander Pope, *An Essay on Criticism*

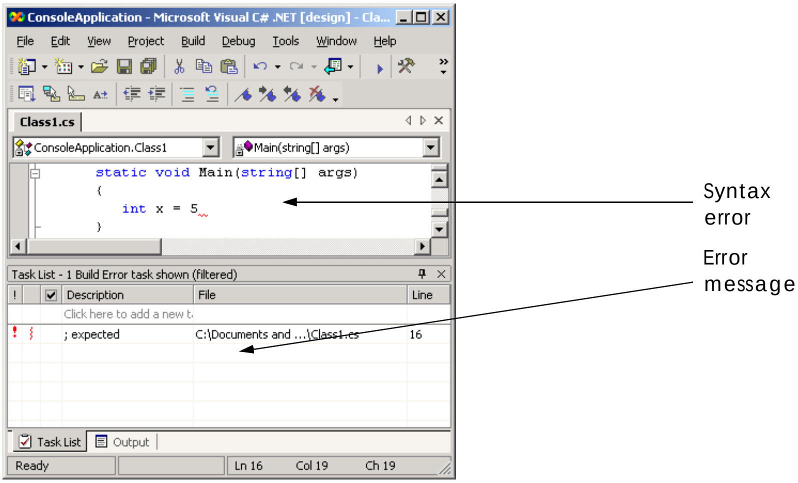
Outline

D.1 Introduction
D.2 Breakpoints
D.3 Examining Data
D.4 Program Control
D.5 Additional Method Debugging Capabilities
D.6 Additional Class Debugging Capabilities
Summary

D.1 Introduction

Two types of errors occur during software development: syntax errors and logic errors. Syntax errors (or compilation errors) occur when program statements violate the grammatical rules of a programming language, such as failure to end a statement with a semicolon. When a compiler detects syntax errors, the compiler terminates without building the application. By contrast, logic errors do not prevent programs from compiling or executing, but rather prevent programs from operating as expected.

Syntax errors are easier to fix than are logic errors. Upon detecting a syntax error, the compiler gives the description and line number in the ***Task List*** window (Fig. D.1). This information gives the programmer a "clue" as to how to eliminate the error, so the compiler can create the program. However, logic errors often are more subtle and usually do not inform the user exactly where in the program the error occurred. This appendix overviews both types of errors and details Visual Studio .NET's capabilities for detecting and correcting the these logic errors.


ConsoleApplication - Microsoft Visual C# .NET [design] - Cla...
File Edit View Project Build Debug Tools Window Help
Class1.cs
ConsoleApplication.Class1
Main(string[] args)
Task List - 1 Build Error task shown (filtered)
Task List
Output
Ready
Ln 16 Col 19 Ch 19
Syntax error
Error message

```
static void Main(string[] args)
{
   int x = 5
}
```

!	☑	Description	File	Line
		Click here to add a new t		
!		; expected	C:\Documents and ...\Class1.cs	16

Fig. D.1 Syntax error.

Testing and Debugging Tip D.1

After fixing one error, you may observe that the number of overall errors perceived by the compiler is significantly reduced.

Testing and Debugging Tip D.2

When the compiler reports a syntax error on a particular line, check that line for the syntax error. If the error is not on that line, check the preceding few lines of code for the cause of the syntax error.

Debugging is the process of finding and correcting *logic errors* in applications. Logic errors are more subtle than syntax errors because a program that includes a logic error compiles successfully but does not run as expected. Logic errors often are difficult to debug, because the programmer cannot see the code as it executes. One strategy that novice programmers often use to debug programs is to display program data directly, using message boxes or `Console.WriteLine` statements. For example, the programmer might print the value of a variable when its value changes to determine whether the variable is assigned the correct value. This approach is cumbersome, because programmers must insert a line of code wherever they suspect there might be a problem. Furthermore, once the program has been debugged, the programmer then must remove the extraneous statements, which often can be difficult to distinguish from the original program code.

A *debugger* is software that allows a programmer to analyze program data and trace the flow of program execution while the application runs. A debugger provides capabilities that allow the programmer to suspend program execution, examine and modify variables, call methods without changing the program code and more. In this appendix, we introduce the Visual Studio .NET debugger and several of its debugging tools. [*Note*: A program must successfully compile before it can be used in the debugger.]

D.2 Breakpoints

Breakpoints are a simple but effective debugging tool. A breakpoint is a marker that a programmer places in a code listing. When a program reaches a breakpoint, execution pauses—this allows the programmer to examine the state of the program and ensure that it is working as expected. Figure D.2 is a program that outputs the value of ten factorial (10!)[1], but contains two logic errors—the first iteration of the loop multiplies `x` by `10` instead of multiplying `x` by `9`, and the result of the factorial calculation is multiplied by `0` (so the result is always `0`). We use this program to demonstrate Visual Studio .NET's debugging abilities—using its breakpoint capabilities as our first example.

```
// Fig. D.2: DebugExample.cs
// Sample program to debug.

using System;

```

Fig. D.2 Debug sample program. (Part 1 of 2.)

---

[1] The factorial of `x` (`x`!) is defined as the product of all digits less than or equal to `x` but greater than zero. For example, 10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1.

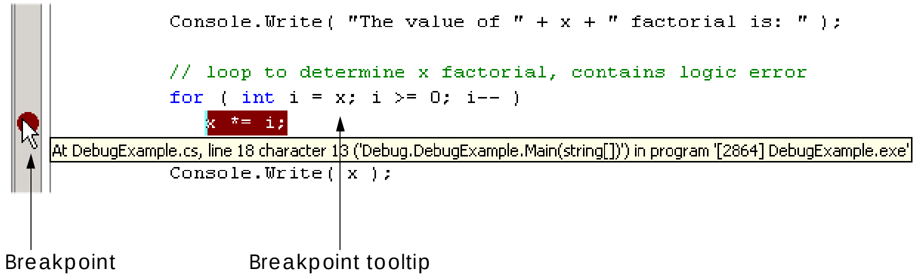
```
namespace Debug
{
   class DebugExample
   {
      static void Main( string[] args )
      {
         int x = 10;

         Console.Write( "The value of " + x + " factorial is: " );

         // loop to determine x factorial, contains logic error
         for ( int i = x; i >= 0; i-- )
            x *= i;

         Console.Write( x );

         Console.ReadLine(); // delay program exit

      } // end main

   } // end class DebugExample

} // end namespace Debug
```

```
The value of 10 factorial is: 0
```

Fig. D.2 Debug sample program. (Part 2 of 2.)

To set breakpoints in Visual Studio, click the gray area to the left of any line of code or right-click a line of code and select **Insert Breakpoint**. A solid red circle appears, indicating that the breakpoint has been set (Fig. D.3). The program execution is suspended when it reaches the line containing the breakpoint.

To enable breakpoints and other debugging features, we must compile the program using the debug configuration (Fig. D.4). Select **Debug** from the configuration toolbar if it is not already selected. Alternatively, select **Build > Configuration Manager** and change the **Active Solution Configuration** to **Debug**.



```
         Console.Write( "The value of " + x + " factorial is: " );

         // loop to determine x factorial, contains logic error
         for ( int i = x; i >= 0; i-- )
            x *= i;

         Console.Write( x );
```

At DebugExample.cs, line 18 character 13 ('Debug.DebugExample.Main(string[])') in program '[2864] DebugExample.exe'

Fig. D.3 Setting a breakpoint.

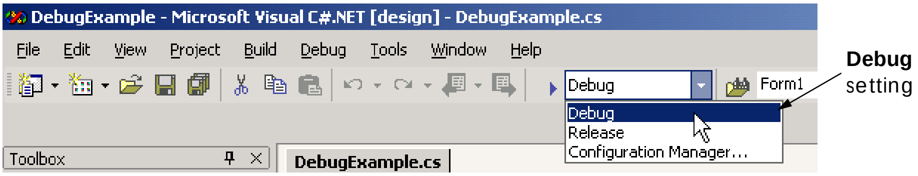

DebugExample - Microsoft Visual C#.NET [design] - DebugExample.cs
File Edit View Project Build Debug Tools Window Help
Debug
Form1
Debug
Release
Configuration Manager...
Toolbox
DebugExample.cs
Debug setting

Fig. D.4 Debug configuration setting.

Selecting **Debug > Start** compiles the program and begins debugging. When debugging a console application, the console window appears (Fig. D.5), allowing program interaction (input and output). When the debugger reaches the breakpoint (line 18) program execution is suspended, and the IDE becomes the active window. Programmers may need to switch between the IDE and the console window while debugging programs.

Figure D.6 shows the IDE with program execution suspended at the breakpoint. The *yellow arrow* to the left of the statement

```
x *= i;
```

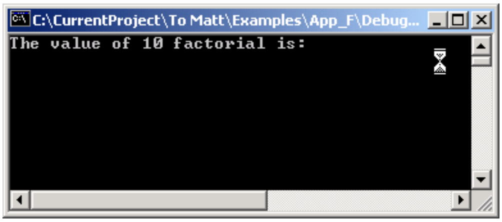
indicates the line at which execution is suspended and that the line contains the next statement to execute. Note that the title bar of the IDE displays **[break]**—this indicates that the IDE is in *break mode* (i.e., the debugger is running). Once the program reaches the breakpoint, a programmer can "hover" with the mouse on a variable (in this case `x` or `i`) in the source code to view the value of that variable in a tooltip as shown in Fig. D.6.

Testing and Debugging Tip D.3

Placing a breakpoint after a loop in a program allows the loop to complete without stopping before the breakpoint is reached.

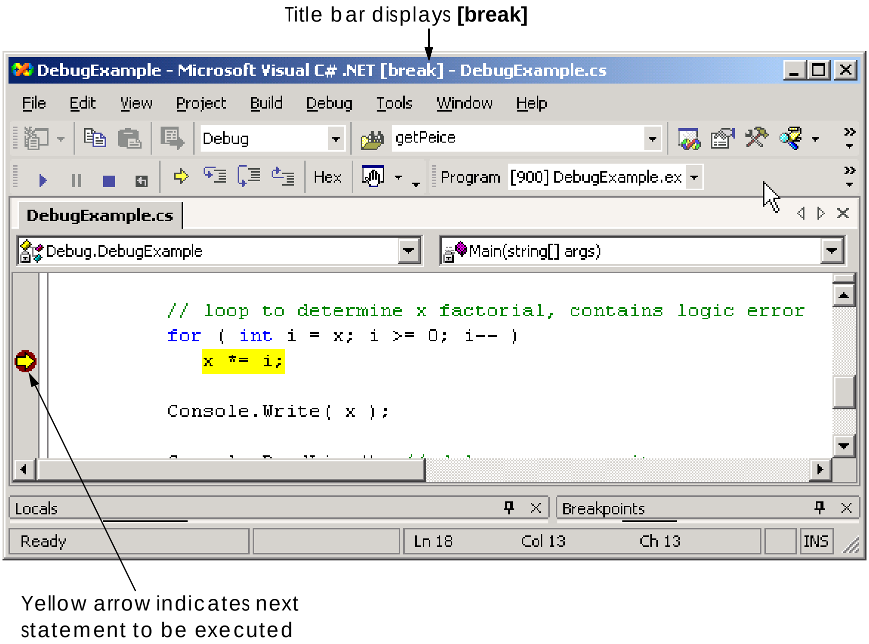
D.3 Examining Data

Visual Studio .NET includes several debugging windows that allow programmers to examine variables and expressions. All the windows are accessible from the **Debug > Windows** submenu. Some windows are listed only when the IDE is in break mode (also called *debug mode*). The ***Watch*** window, which is available only in break mode (Fig. D.7), allows programmers to examine the values of related groups of variables and expressions. Visual Studio .NET provides a total of four **Watch** windows.


C:\CurrentProject\To Matt\Examples\App_F\Debug...

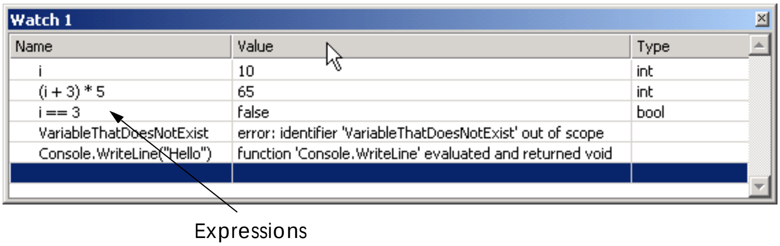
```
The value of 10 factorial is:
```

Fig. D.5 Console application suspended for debugging.


Title bar displays **[break]**
DebugExample - Microsoft Visual C# .NET [break] - DebugExample.cs
File Edit View Project Build Debug Tools Window Help
Debug
getPeice
Hex
Program [900] DebugExample.ex
DebugExample.cs
Debug.DebugExample
Main(string[] args)
Locals
Breakpoints
Ready
Ln 18
Col 13
Ch 13
INS
Yellow arrow indicates next statement to be executed

```
// loop to determine x factorial, contains logic error
for ( int i = x; i >= 0; i-- )
   x *= i;

Console.Write( x );
```

Fig. D.6 Execution suspended at a breakpoint.


Watch 1
Expressions

Name	Value	Type
i	10	int
(i + 3) * 5	65	int
i == 3	false	bool
VariableThatDoesNotExist	error: identifier 'VariableThatDoesNotExist' out of scope	
Console.WriteLine("Hello")	function 'Console.WriteLine' evaluated and returned void	

Fig. D.7 **Watch** window.

Upon first opening, the **Watch** window will not contain any expressions to evaluate. To examine data, type an expression into the ***Name*** field. Most valid C# expressions can be entered in the **Name** field, including expressions that contain method calls. Consult the documentation under "**debugger, expressions**" for a full description of valid expressions.

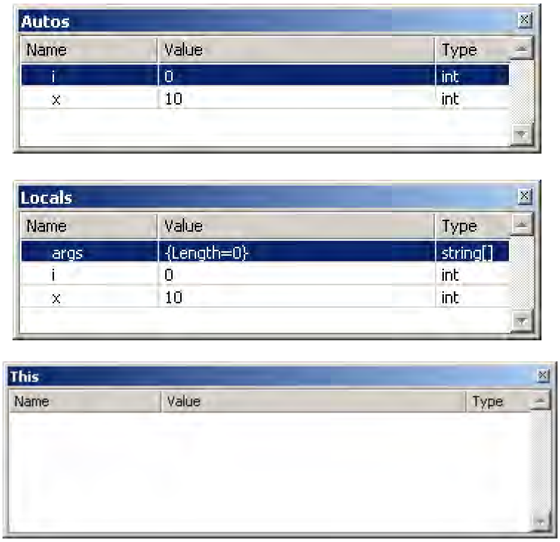
Once an expression is entered, its type and value appear in the ***Value*** and ***Type*** fields. The first expression entered is the variable **`i`**, which has a value of **`10`** (line 12 assigns the value of **`10`** to variable **`x`**, and line 17 assigns the value of **`x`** to **`i`**). The **Watch** window also can evaluate more complex arithmetic expressions (e.g, **`(i + 3) * 5`**). Thus, the **Watch** window provides a convenient way to display various types of program data without modifying code.

By entering the variables and expressions that are relevant to a program's logic error, programmers can trace incorrect values to the source of the error and eliminate it. For example, to debug the program in Fig. D.2, we might enter the expression `i * x` in the **Watch** window. When we reach the breakpoint for the first time, the expression has a value `100` instead of `90`, which indicates a logic error in our program. This occurs because the loop at lines 17–18 started multiplying `x` by `10` as opposed to multiplying by `9`. We subtract `1` from the initial value that the `for` loop assigns to `i` (i.e., change `10` to `9`) to correct the error.

If a **Name** field in the **Watch** window contains a variable name, the variable's value can be modified for debugging purposes. To modify a variable's value, click its value in the **Value** field and enter a new value. Any modified value appears in red.

If an expression is invalid, an error appears in the **Value** field. For example, **VariableThatDoesNotExist** is not an identifier used in the program (fourth line in Fig. D.7). Therefore, Visual Studio .NET issues an error message in the **Value** field. To remove an expression, select it and press *Delete*.

Visual Studio also provides the ***Locals***, ***Autos*** and ***This*** windows (Fig. D.8), which are similar to the **Watch** window, except the programmer does not specify their contents. The **Locals** window displays the name and current value for all the variables that have block scope in the method containing the current statement (indicated by the yellow arrow in Fig. D.6). The **Autos** window displays the variables and values of the current statement and the previous statement. Variables can be changed in either window by clicking the appropriate **Value** field and entering a new value. The **This** window displays data that has class scope for an object. If the program is inside a `static` method (such as method `Main` in a console application), the **This** window is empty.



Autos

Name	Value	Type
i	0	int
x	10	int

Locals

Name	Value	Type
args	{Length=0}	string[]
i	0	int
x	10	int

This

Name	Value	Type

Fig. D.8 **Autos**, **Locals** and **This** windows.

A programmer can evaluate expressions line-by-line in the ***Immediate*** window (Fig. D.9). To evaluate an expression, a programmer types this expression into the window and presses *Enter*. For example, when a programmer enters `Console.WriteLine(i)` and presses *Enter*, the value of `i` is output to the console window. A developer also can use the assignment operator (`=`) to perform assignments in the **Immediate** window. Notice that the values for `i` and `x` in the **Locals** window contain these updated values.

Testing and Debugging Tip D.4

Use the ***Immediate*** *window to call a method one time. Placing a method call inside the* ***Watch*** *window calls that method every time the program breaks.*

D.4 Program Control

The Visual Studio .NET Debugger give programmers considerable control over the execution of a program. Using breakpoints and program-control commands provided by the debugger, programmers conveniently can analyze the execution of code at any point in a program. This is useful when a program contains multiple calls to methods that are known to execute properly. The **Debug** toolbar contains buttons that provide convenient access for controlling the debugging process (Fig. D.10). To display the **Debug** toolbar, select **View > Toolbars > Debug**.

The debug toolbar in Fig. D.10 controls debugger execution. The ***Restart*** button executes the program from the beginning, pausing at the beginning of the program to allow the programmer to set breakpoints before the program executes again. The ***Continue*** button resumes execution of a suspended program. The ***Stop Debugging*** button ends the debugging session, and the ***Break All*** button allows the programmer to suspend an executing program directly (i.e., without explicitly setting breakpoints). After execution suspends, the yellow arrow appears indicating the next statement to be executed.

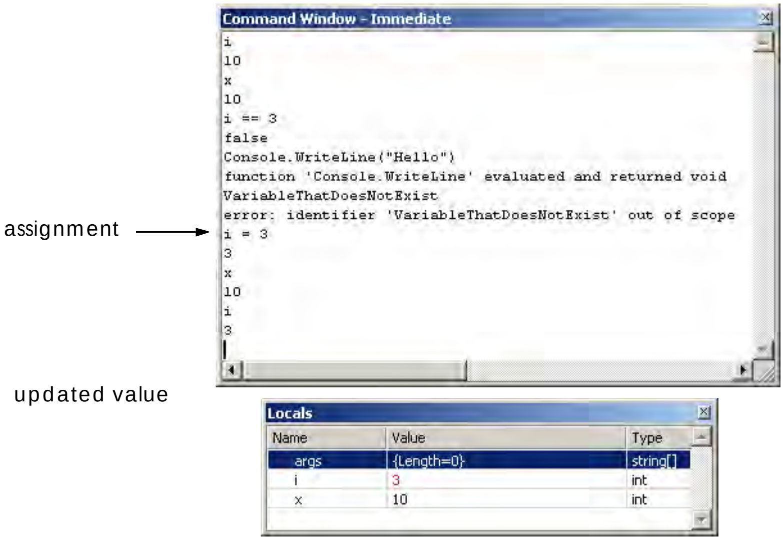
Testing and Debugging Tip D.5

When a program is executing, problems such as infinite loops usually can be interrupted by selecting ***Debug > Break All*** *or by clicking the corresponding button on the toolbar.*

Clicking the ***Show Next Statement*** button places the cursor on the same line as the yellow arrow. This command is useful when a programmer needs to return to the current execution point after setting breakpoints in a program that contains many lines of code.

The ***Step Over*** button executes the next executable statement and advances the yellow arrow to the following line. If the next line of code contains a method call, the method is executed in its entirety as one step. This button allows the user to execute the program one line at a time without seeing the details of every method that is called. This is useful when a program contains multiple calls to methods that are known to execute properly. We discuss the **Step Into** and **Step Out** buttons in the next section.

The **Hex** button toggles the display format of data. If enabled, **Hex** displays data in hexadecimal (base 16) format, rather than displaying data in decimal (base 10) format. Experienced programmers often prefer to read values in hexadecimal format—especially large numbers because hexadecimal number representation is more concise and can be converted easily to binary (base 2) form. For more information about the hexadecimal and decimal number formats, see Appendix B, Number Systems.


Command Window - Immediate
Locals
assignment
updated value

```
i
10
x
10
i == 3
false
Console.WriteLine("Hello")
function 'Console.WriteLine' evaluated and returned void
VariableThatDoesNotExist
error: identifier 'VariableThatDoesNotExist' out of scope
i = 3
3
x
10
i
3
```

Name	Value	Type
args	{Length=0}	string[]
i	3	int
x	10	int

Fig. D.9 **Immediate** window.

The **Breakpoints** window displays all the breakpoints set for the program (Fig. D.11). A checkbox appears next to each breakpoint, indicating whether the breakpoint is *active* (checked) or *disabled* (unchecked). Lines with disabled breakpoints contain an unfilled red circle rather than a solid one (Fig. D.12). The debugger does not pause execution at disabled breakpoints.

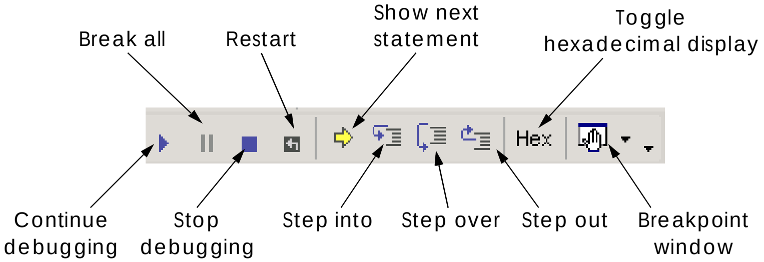
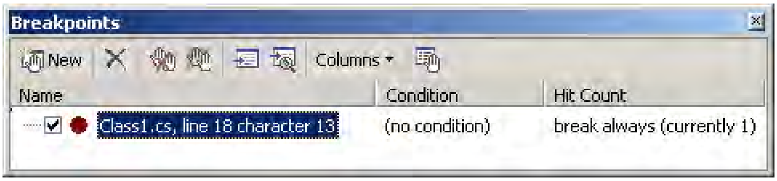
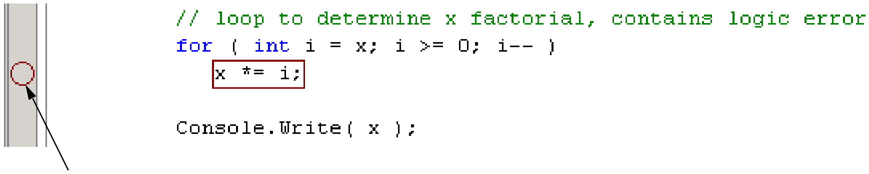

Break all
Restart
Show next statement
Toggle hexadecimal display
Hex
Continue debugging
Stop debugging
Step into
Step over
Step out
Breakpoint window

Fig. D.10 **Debug** toolbar icons.


Breakpoints
New
Columns

Name	Condition	Hit Count
☑ ● Class1.cs, line 18 character 13	(no condition)	break always (currently 1)

Fig. D.11 **Breakpoints** window.



```
// loop to determine x factorial, contains logic error
for ( int i = x; i >= 0; i-- )
   x *= i;

Console.Write( x );
```

Disabled breakpoint

Fig. D.12 Disabled breakpoint.

Testing and Debugging Tip D.6

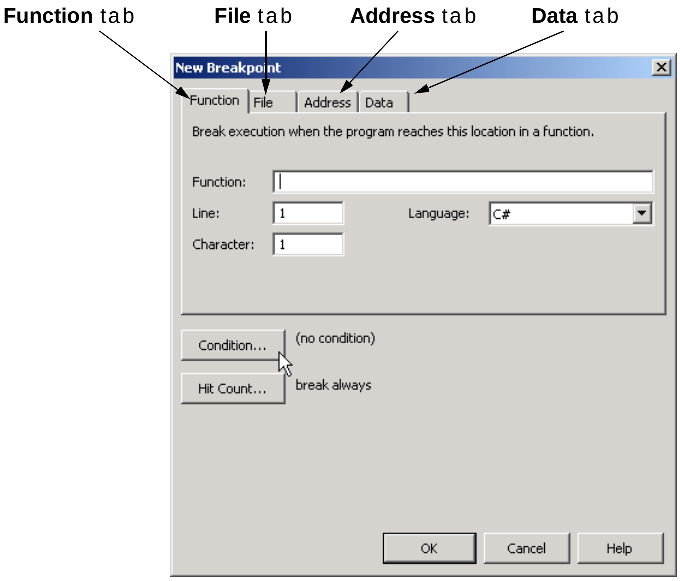
Disabled breakpoints allow the programmer to maintain breakpoints in key locations in the program so they can be reactivated when needed. Disabled breakpoints are always visible.

In the **Breakpoints** window (Fig. D.11), the ***Condition*** field displays the condition that must be satisfied to suspend program execution at that breakpoint. The ***Hit Count*** field displays the number of times the debugger has stopped at each breakpoint. Double-clicking an item in the **Breakpoints** window moves the cursor to the line containing that breakpoint.

A programmer can add breakpoints to a program by clicking the ***New*** button in the **Breakpoints** window. This causes a **New Breakpoint** dialog to display (Fig. D.13). The **Function**, **File**, **Address** and **Data** tabs allow the programmer to suspend execution at either a method, a line in a particular file, an instruction in memory or when the value of a variable changes. The **Hit Count...** button (Fig. D.14) can be used to specify when the breakpoint should suspend the program (the default is to **`always break`**). A breakpoint can be set to suspend the program when the hit count reaches a specific number, when the hit count is a multiple of a number or is greater than or equal to a specific number.

The Visual Studio debugger also allows execution to suspend at a breakpoint depending on the value of an expression. Clicking the **Condition…** button opens the ***Breakpoint Condition*** dialog (Fig. D.15). The **Condition** checkbox indicates whether breakpoint conditions are enabled. The radio buttons determine how the expression in the text box is evaluated. The **is true** radio button pauses execution at the breakpoint whenever the expression is true. The **has changed** radio button causes program execution to suspend when it first encounters the breakpoint and again each time the expression differs from its previous value when the breakpoint is encountered. When the **New Breakpoint** dialog has been closed, the **Breakpoints** window displays the condition and hit count options for the new break point.

Suppose we set **`x * i != 0`** as the condition for the breakpoint in our loop, with the **has changed** option enabled. (We might choose to do this because the program produces an incorrect output of `0`). Program execution suspends when it first reaches the breakpoint and records that the expression has a value of **`true`**, because **`x * i`** is `100` (or `10` if we fixed the earlier logic error). We continue, and the loop decrements **`i`**. While **`i`** is between `10` and `1`, the condition's value never changes, and execution is not suspended at that breakpoint. When **`i`** is `0`, the expression **`x * i != 0`** is **`false`**, and execution is suspended. At this point, the programmer identifies the second logic error in our program—the final iteration of the **`for`** loop multiplies the result by `0`. To return the IDE to design mode, click the **Stop Debugging** button on the **Debug** toolbar.


Function tab
File tab
Address tab
Data tab
New Breakpoint
Function
File
Address
Data
Condition...
Hit Count...
OK
Cancel
Help

Break execution when the program reaches this location in a function.

Function:

Line: 1

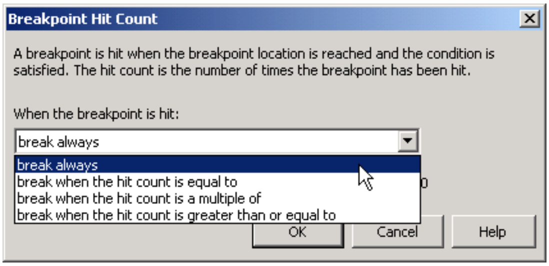
Language: C#

Character: 1

(no condition)

break always

Fig. D.13 **New Breakpoint** dialog.


Breakpoint Hit Count
OK
Cancel
Help

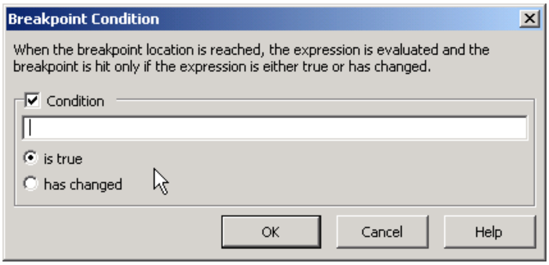
A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

When the breakpoint is hit:

break always

- break always
- break when the hit count is equal to
- break when the hit count is a multiple of
- break when the hit count is greater than or equal to

Fig. D.14 **Breakpoint Hit Count** dialog.


Breakpoint Condition
OK
Cancel
Help

When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is either true or has changed.

☑ Condition

● is true

○ has changed

Fig. D.15 **Breakpoint Condition** dialog.

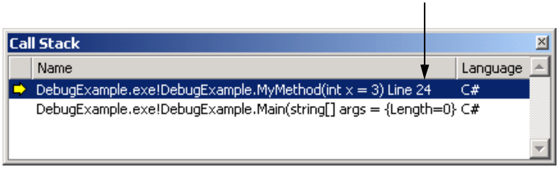
D.5 Additional Method Debugging Capabilities

In programs with many methods, it is often difficult to determine which methods may have been involved in incorrect calculations that resulted in a logic error. To simplify this process, the Visual Studio debugger includes tools for analyzing methods and method calls. We demonstrate some method-debugging tools in the following example (Fig. D.16).

The ***Call Stack*** window contains the program’s *method call stack*, which allows the programmer to determine the exact sequence of calls that lead to the current method and to examine calling methods on the stack. This window allows the programmer to determine the flow of control in the program that resulted in the execution of the current method. For example, a breakpoint is inserted in **`MyMethod`**, the call stack in (Fig. D.17) indicates that the program called method **`Main`** first, followed by **`MyMethod`**.

```
// Fig. D.16: MethodDebugExample.cs
// Demonstrates debugging methods.

using System;

namespace Debug
{

   // provides methods on which to demonstrate
   // Visual Studio’s debug tools
   class MethodDebug
   {
      // entry point for application
      static void Main( string[] args )
      {
         // display MyMethod return values
         for ( int i = 0; i < 10; i++ )
            Console.WriteLine( MyMethod( i ) );

         Console.ReadLine();
      } // end method main

      // perform calculation
      static int MyMethod( int x )
      {
         return ( x * x ) - ( 3 * x ) + 7;
      } // end method MyMethod

      // method with logic error
      static int BadMethod( int x )
      {
         return 1 / ( x - x );
      } // end method BadMethod

   } // end class MethodDebug

} // end namespace Debug
```

Fig. D.16 Debugging methods.



Most recently called method

	Name	Language
➪	DebugExample.exe!DebugExample.MyMethod(int x = 3) Line 24	C#
	DebugExample.exe!DebugExample.Main(string[] args = {Length=0})	C#

Fig. D.17 **Call Stack** window.

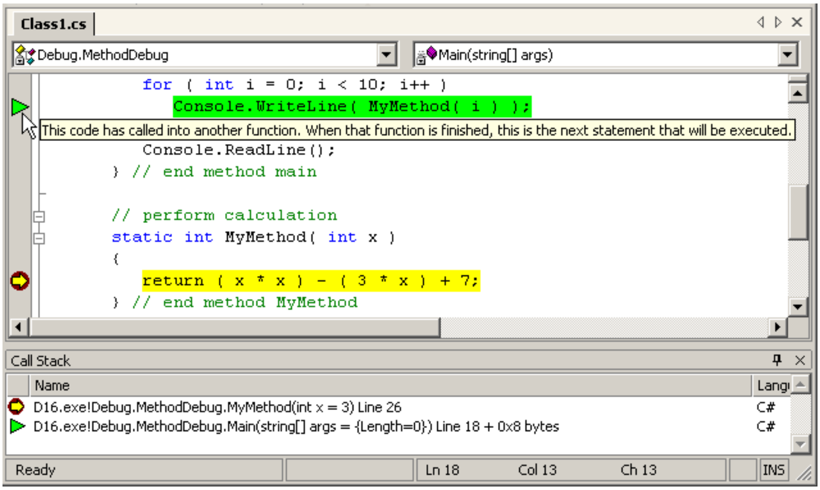
Double-clicking any line in the **Call Stack** window displays the next line to be executed in that method. This allows the programmer to determine how the result of each method will affect the calling method's execution. Visual Studio .NET highlights the line in green and displays the tooltip shown in Fig. D.18.

Visual Studio .NET also provides additional program-control buttons for debugging methods. The **Step Over** button executes one statement in a method, then pauses program execution at the following line. Using **Step Over**, if an evaluated statement invokes a method, the method is invoked, and execution stops at the next statement. Using ***Step Into***, if a statement invokes a method, control transfers to the method for line-by-line. The ***Step Out*** button finishes executing the current method and returns control to the line that called the method.

Testing and Debugging Tip D.7

*Use **Step Out** to finish a method that was stepped into accidentally.*

Figure D.19 lists each program-control debug feature, its shortcut key and a description. Experienced programmers often prefer using these shortcut keys to access menu commands.



```
            for ( int i = 0; i < 10; i++ )
               Console.WriteLine( MyMethod( i ) );
            Console.ReadLine();
         } // end method main

         // perform calculation
         static int MyMethod( int x )
         {
            return ( x * x ) - ( 3 * x ) + 7;
         } // end method MyMethod
```

This code has called into another function. When that function is finished, this is the next statement that will be executed.

	Name	Lang
○	D16.exe!Debug.MethodDebug.MyMethod(int x = 3) Line 26	C#
▷	D16.exe!Debug.MethodDebug.Main(string[] args = {Length=0}) Line 18 + 0x8 bytes	C#

Fig. D.18 IDE displaying a method's calling point.

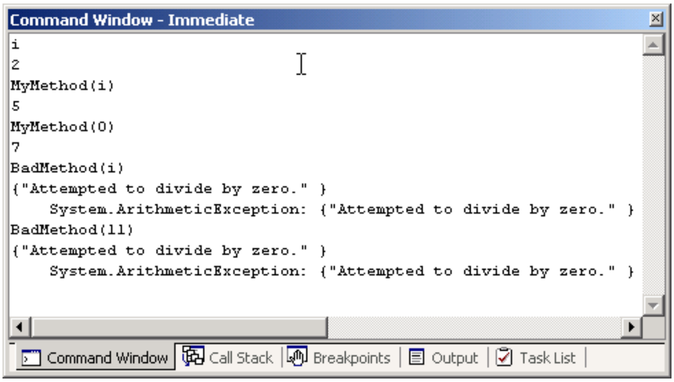
Control Button	Shortcut Key	Description
Continue	*F5*	Continues program execution. Execution continues until either a breakpoint is encountered or the program ends (through normal execution).
Stop Debugging	*Shift + F5*	Stops debugging and returns to Visual Studio design mode.
Step Over	*F10*	Advances to next statement, does not step into method calls.
Step Into	*F11*	Executes next statement. If the statement contains a method call, control transfers to the method for line-by-line debugging. If the statement does not contain a method call, **Step Into** behaves like **Step Over**.
Step Out	*Shift + F11*	Finishes executing the current method and suspends program execution in the calling method.

Fig. D.19 Debug program control features.

Programmers can use the **Immediate** window, discussed in Section D.3 for testing method arguments passed to a method (Fig. D.20). Testing the arguments helps determine if a method is functioning properly.

D.6 Additional Class Debugging Capabilities

In most sophisticated C# programs, a large portion of program data is contained in objects. For these purposes, Visual Studio includes class debugging features, which allow programmers to determine the current state of objects used in a program. We demonstrate some class debugging features using the code presented in Fig. D.21. To examine an instance of class **`DebugEntry`**, we place a breakpoint at line 43, as shown in Fig. D.22. [*Note*: A C# file may contain multiple classes, as is the case with this example.]



```
i
2
MyMethod(i)
5
MyMethod(0)
7
BadMethod(i)
{"Attempted to divide by zero." }
    System.ArithmeticException: {"Attempted to divide by zero." }
BadMethod(11)
{"Attempted to divide by zero." }
    System.ArithmeticException: {"Attempted to divide by zero." }
```

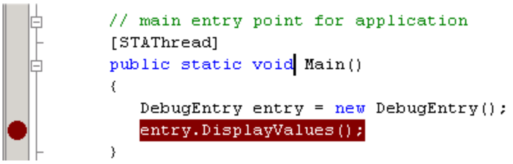
Fig. D.20 Using the **Immediate** window to debug methods.

```
// Fig. D.21: DebugClass.cs
// Console application to demonstrate object debugging.

using System;

namespace ClassDebug
{

   // creates array containing three different classes
   public class DebugEntry
   {
      public int someInteger = 123;
      private int[] integerArray = { 74, 101, 102, 102 };
      private DebugClass debugClass;
      private Random randomObject;
      private object[] list = new object[ 3 ];

       // constructor
       public DebugEntry()
       {
          randomObject = new Random();
          debugClass = new DebugClass( "Hello World",
            new object() );

          list[ 0 ] = integerArray;
          list[ 1 ] = debugClass;
          list[ 2 ] = randomObject;
       }

       // display values retrieved from three objects
       public void DisplayValues()
       {
         Console.WriteLine( randomObject.Next() );
         Console.WriteLine( debugClass.SomeString );
         Console.WriteLine( integerArray[ 0 ] );
       }

      // main entry point for application
      [STAThread]
      public static void Main()
      {
         DebugEntry entry = new DebugEntry();
         entry.DisplayValues();
      }
   } // end class DebugEntry

   // demonstrates class debugging
   public class DebugClass
   {
      // private variables
      private string someString;
      private object privateObject;

```

Fig. D.21 Object debugging example. (Part 1 of 2.)

```
      // constructor
      public DebugClass( string stringData,
         object objectData )
      {
         someString = stringData;
         privateObject = objectData;
      }

      // accessor property for someString
      public string SomeString
      {
         get
         {
           return someString;
         }

         set
         {
            someString = value;
         }
      } // end property SomeString

   } // end class DebugClass

} // end namespace ClassDebug
```

Fig. D.21 Object debugging example. (Part 2 of 2.)



```
// main entry point for application
[STAThread]
public static void Main()
{
   DebugEntry entry = new DebugEntry();
   entry.DisplayValues();
}
```

Fig. D.22 Breakpoint location for class debugging.

To assist class debugging, Visual Studio .NET allows the programmer to expand and view all data members and properties of a class, including **`private`** members. In any of the three windows (i.e., **Watch**, **Locals**, **Autos** and **This**), a class that has data members is displayed with a plus (**+**) (Fig. D.23). When a programmer clicks the plus box, all the object's data members and their values display. If a member references an object, the object's data members also can be listed by clicking the object's plus box.

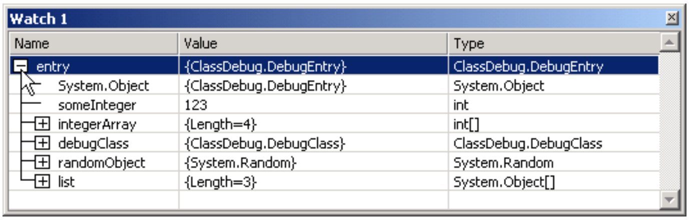
Many logic errors are the result of incorrect array calculations. To simplify the identification of such errors, the debugger includes the ability to display all the values in an array. Figure D.24 displays the contents of the **`list`** array. The object at index **`0`** is and **`int`** array, which is expanded to show its contents. Index **`1`** contains a **`DebugClass`** object—expanded to show the object's **`private`** data members, as well as a **`public`** property. Index **`2`** contains a **`Random`** object, defined in the Framework Class Library (FCL).

The Visual Studio debugger contains several other debugging windows, including **Threads**, **Modules**, **Memory**, **Disassembly** and **Registers**. These windows are used by experienced programmers to debug large, complex projects—consult the Visual Studio .NET documentation for more details on these features.

In this appendix we demonstrated several techniques for debugging programs, methods and classes. The Visual Studio .NET debugger is a powerful tool, which allows programmers to build more robust, fault-tolerant programs.

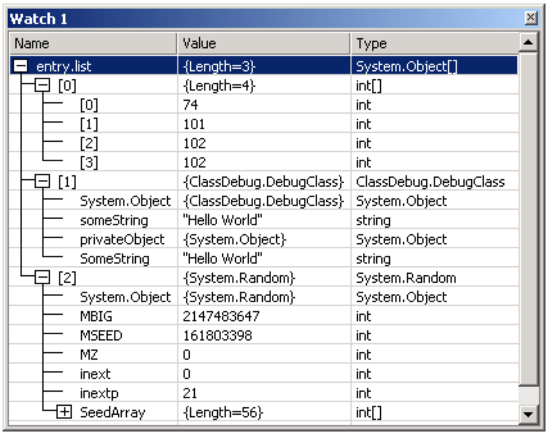
SUMMARY

- Debugging is the process of finding logic errors in applications.
- Syntax errors (or compilation errors) occur when program statements violate the grammatical rules of a programming language. These errors are caught by the compiler.
- Logic errors are more subtle than syntax errors. They occur when a program compiles successfully, but does not run as expected.
- Debuggers can suspend a program at any point, which allows programmers to examine and set variables and call methods.
- A breakpoint is a marker set at a line of code. When a program reaches a breakpoint, execution is suspended. The programmer then can examine the state of the program and ensure that the program is working properly.



Name	Value	Type
entry	{ClassDebug.DebugEntry}	ClassDebug.DebugEntry
System.Object	{ClassDebug.DebugEntry}	System.Object
someInteger	123	int
integerArray	{Length=4}	int[]
debugClass	{ClassDebug.DebugClass}	ClassDebug.DebugClass
randomObject	{System.Random}	System.Random
list	{Length=3}	System.Object[]

Fig. D.23 Expanded class in **Watch** window.



Name	Value	Type
entry.list	{Length=3}	System.Object[]
[0]	{Length=4}	int[]
[0]	74	int
[1]	101	int
[2]	102	int
[3]	102	int
[1]	{ClassDebug.DebugClass}	ClassDebug.DebugClass
System.Object	{ClassDebug.DebugClass}	System.Object
someString	"Hello World"	string
privateObject	{System.Object}	System.Object
SomeString	"Hello World"	string
[2]	{System.Random}	System.Random
System.Object	{System.Random}	System.Object
MBIG	2147483647	int
MSEED	161803398	int
MZ	0	int
inext	0	int
inextp	21	int
SeedArray	{Length=56}	int[]

Fig. D.24 Expanded array in **Watch** window.

- To enable the debugging features, the program must be compiled using the debug configuration.
- To set breakpoints, click the gray area to the left of any line of code. Alternatively, right-click a line of code and select **Insert Breakpoint**.
- The **Watch** window allows the programmer to examine variable values and expressions. To examine data, type a valid Visual Basic expression, such as a variable name, into the **Name** field. Once the expression has been entered, its type and value appear in the **Type** and **Value** fields.
- Variables in the **Watch** window can be modified by the user for testing purposes. To modify a variable's value, click the **Value** field and enter a new value.
- The **Locals** window displays the name and current value for all the local variables or objects in the current scope.
- The **Autos** window displays the variables and objects used in the previous statement and the current statement (indicated by the yellow arrow).
- To evaluate an expression in the **Immediate** window, simply type the expression into the window and press *Enter*.
- The **Continue** button resumes execution of a suspended program.
- The **Stop Debugging** button ends the debugging session.
- The **Break All** button allows the programmer to place an executing program in break mode.
- The **Show Next Statement** button places the cursor on the same line as the yellow arrow that indicates the next statement to execute.
- The **Step Over** button executes the next executable line of code and advances the yellow arrow to the following executable line in the program. If the line of code contains a method call, the method is executed in its entirety as one step.
- The **Hex** button toggles the display format of data. If enabled, **Hex** displays data in a hexadecimal (base 16) form, rather than decimal (base 10) form.
- The **Breakpoints** window displays all the breakpoints currently set for a program.
- Disabled breakpoints allow the programmer to maintain breakpoints in key locations in the program so they can be used again when needed.
- The **Call Stack** window contains the program's method call stack, which allows the programmer to determine the exact sequence of calls that led to the current method and to examine calling methods on the stack.
- The **Step Over** button executes one statement in a method, then pauses program execution.
- The **Step Into** button executes next statement. If the statement contains a method call, control transfers to the method for line-by-line debugging. If the statement does not contain a method call, **Step Into** behaves like **Step Over**.
- The **Step Out** finishes executing the method and returns control to the line that called the method.
- The **Immediate** window is useful for testing arguments passed to a method. This helps determine if a method is functioning properly.
- Visual Studio .NET includes class debugging features which allow the programmer to determine the current state of any objects used in a program.
- To assist class debugging, Visual Studio .NET allows the programmer to expand and view all data members variables and properties of an object, including those declared `private`.

E

Generating Documentation in Visual Studio

Objectives

- To introduce Visual Studio .NET's documentation generation tool.
- To introduce XML documentation comments.
- To understand XML documentation tags and their use.
- To be able to generate HTML and XML documentation files.

Outline

E.1 Introduction
E.2 Documentation Comments
E.3 Documenting C# Source Code
E.4 Creating Comment Web Pages
E.5 Creating XML Documentation Files

Terminology • Summary

E.1 Introduction

A single programmer can implement most of the programs from this book. However, industrial software is more complex, and each project almost always requires the talent of several programmers. In such projects, communication among programmers is a necessity. When a programmer writes code for a class, programmers from the same group should understand how that class operates. For this reason, each programmer should document specific information on a class, such as the class's role in a system, the functionality that each method provides for the class and the purpose of each class variable. This documentation helps all programmers understand how classes can interoperate, and facilitates modification, use and extension of each class.

To facilitate the creation of documentation for a project, Visual Studio .NET provides the *XML documentation tool*. This tool converts key pieces of information in source code—such as the class's members, the hierarchy to which the class belongs and any other general remarks the programmer wishes to document—to HTML[1] or XML[2] format. The programmer specifies the general remarks to be documented by placing them in special regions in the code, called *XML documentation comments*.

In this appendix, we introduce Visual Studio .NET's documentation capabilities. We begin by discussing the format and structure of the XML documentation comments that the documentation-generation tool uses to create the documentation files. We then show how to generate the documentation through a LIVE-CODE™ example. We recommend reading through Chapters 8–10 before reading this appendix, because the examples presented in this appendix relate to the examples from these chapters.

E.2 Documentation Comments

Before the Visual Studio documentation-generation tool can generate documentation files, the programmer must insert XML documentation comments into the source files. These comments contain the information that the programmer wishes to document. The documentation-generation tool recognizes only single-line comments that begin with three forward slashes (`///`). An example of a simple documentation comment is

```
/// <summary>
```

---

[1] HTML is discussed in Appendices I and J.
[2] XML is discussed in Chapter 18.

```
/// this is a comment
/// </summary>
```

In this comment, the first line begins element **`summary`**, the second line defines the text that element **`summary`** contains and the third line closes element **`summary`**. As we will discuss later in this text, the documentation tool will document only the text within these **`summary`** tags. All XML declaration comments (excluding the three forward slashes) must contain well-formed XML. Like general comments, the compiler does not translate documentation comments to MSIL (Microsoft Intermediate Language), so they do not become “part of” the program.

Because the documentation tool creates XML files, documentation comments can contain certain types of markup, such as HTML tags and customized XML content. For example, the documentation comment

```
/// <summary>
/// Sorts integer array using <em>MySort</em> algorithm
/// </summary>
```

contains the HTML tags **`<em>`** and **`</em>`**. In the generated HTML files, **`MySort`** appears as *emphasized text* (normally italic).

E.3 Documenting C# Source Code

Figure E.1, Fig. E.2 and Fig. E.3 present a modified version of the **`Point`**, **`Circle`** and **`CircleTest`** classes from Section 9.4 that contains XML documentation comments. In the text that follows the example, we discuss each XML element used in the documentation comments. In Section E.4, we discuss how the documentation tool can generate XML documentation from this file.

```
// Fig. E.1: Point.cs
// Class Point maintains an X and Y coordinate.

using System;

namespace CircleTest
{
   /// <summary>
   /// Class <c><b>Point</b></c> defines a point as a pair
   /// of x and y coordinates.
   /// </summary>
   public class Point
   {
      /// <summary>
      /// This private member of <c><b>Point</b></c>
      /// represents the x coordinate.
      /// </summary>
      /// <returns> The x coordinate as an integer.</returns>
      private int x;
```

Fig. E.1 **`Point`** marked up with XML comments. (Part 1 of 3.)

```

      /// <summary>
      /// This private member of <c><b>Point</b></c>
      /// represents the x coordinate.
      /// </summary>
      /// <returns> The y coordinate as an integer.</returns>
      private int y;

      /// <summary>
      /// Default constructor for class <c><b>Point</b></c>.
      /// </summary>
      /// <remarks>
      /// Sets properties <c><b>X</b></c> and <c><b>Y</b></c> to 0.
      /// </remarks>
      public Point()
      {
         // implicit call to base-class constructor occurs here
      }

      /// <summary>
      /// Constructor for <c><b>Point</b></c> that accepts two
      /// integers that represent the x- and
      /// y coordinates of the point.
      /// </summary>
      /// <remarks>
      /// Uses <c><b>X</b></c> and <c><b>Y</b></c>
      /// properties to set the coordinates of the point,
      /// <em>not</em> private members <c><b>x</b></c>
      /// and <c><b>y</b></c>.
      /// </remarks>
      /// <param name="xValue">
      /// The x coordinate of the circle
      /// </param>
      /// <param name="yValue">
      /// The y coordinate of the circle.
      /// </param>
      public Point( int xValue, int yValue )
      {
         // implicit call to base-class constructor occurs here
         X = xValue;
         Y = yValue;
      }

      /// <summary>
      /// Provides get and set access to member
      /// <c><b>x</b></c>.
      /// </summary>
      /// <value>
      /// <c><b>X</b></c> accesses the value of the
      /// <c><b>x</b></c> data member.
      /// </value>
      public int X
      {
```

Fig. E.1 `Point` marked up with XML comments. (Part 2 of 3.)

```
            get
            {
               return x;
            }

            set
            {
               x = value;
            }
         }

         /// <summary>
         /// Provides get and set access to member
         /// <c><b>y</b></c>.
         /// </summary>
         /// <value>
         /// <c><b>Y</b></c> accesses the value of the
         /// <c><b>y</b></c> data member.
         /// </value>
         public int Y
         {
            get
            {
               return y;
            }

            set
            {
               y = value;
            }
         }

         /// <summary>
         /// Converts the <c><b>Point</b></c> to
         /// <b>string</b> format.
         /// </summary>
         /// <returns>
         /// Returns a string in format:
         /// [x coordinate, y coordinate].
         /// </returns>
         public override string ToString()
         {
            return "[" + X + ", " + Y + "]";
         }

      } // end class Point
   }
```

Fig. E.1 **`Point`** marked up with XML comments. (Part 3 of 3.)

```
   // Fig. E.2: Circle.cs
   // Class Circle inherits from Point.
```

Fig. E.2 **`Circle`** class marked up with XML comments. (Part 1 of 4.)

```

using System;

namespace CircleTest
{
   /// <summary>
   /// Class <c><b>Circle</b></c> inherits from class
   /// <c><b>Point</b></c>. It has an additional member to
   /// represent the radius, a property that provides access
   /// to it and method <c><b>Area</b></c> to compute the area
   /// of the circle.
   /// </summary>
   public class Circle : Point
   {
      /// <summary>
      /// This private member of <c><b>Circle</b></c>
      /// represents the radius.
      /// </summary>
      private double radius;

      /// <summary>
      /// Default constructor for class <c><b>Circle</b></c>.
      /// </summary>
      /// <remarks>
      /// Sets the radius to 0.
      /// </remarks>
      public Circle()
      {
         // implicit call to base class constructor occurs here
      }

      /// <summary>
      /// Constructor for <c>Circle</c> that accepts two integers
      /// that represent the x- and y-coordinates of the circle
      /// and a <b>double</b> that represents the radius.
      /// </summary>
      /// <remarks>
      /// Uses property <c><b>Radius</b></c> to set the radius
      /// of the circle, <em>not</em> private member
      /// <c><b>radius</b></c>.
      /// </remarks>
      /// <param name="xValue">
      /// The x-coordinate of the circle
      /// </param>
      /// <param name="yValue">
      /// The y-coordinate of the circle.
      /// </param>
      /// <param name="radiusValue">
      /// The radius of the circle.
      /// </param>
      public Circle( int xValue, int yValue, double radiusValue )
         : base( xValue, yValue )
      {
```

Fig. E.2 `Circle` class marked up with XML comments. (Part 2 of 4.)

```
            Radius = radiusValue;
         }

         /// <summary>
         /// Provides get and set access to member
         /// <c><b>radius</b></c>.
         /// </summary>
         /// <remarks>
         /// The <c><b>set</b></c> method ensures
         /// that <c><b>radius</b></c> is
         /// <em>not</em> set to a
         /// negative number.
         /// </remarks>
         /// <value>
         /// <c><b>Radius</b></c> accesses the value of the
         /// <c><b>radius</b></c> data member.
         /// </value>
         public double Radius
         {
            get
            {
               return radius;
            }

            set
            {
               if ( value >= 0 )
                  radius = value;
            }
         }

         /// <summary>
         /// Computes the diameter of the circle.
         /// </summary>
         /// <returns>
         /// Returns the diameter of the circle.
         /// </returns>
         public double Diameter()
        {
            return Radius * 2;
         }

         /// <summary>
         /// Computes the circumference of the circle.
         /// </summary>
         /// <remarks>
         /// Uses constant <c><b>Math.PI</b></c>
         /// <see cref="System.Math.PI"/>
         /// </remarks>
         /// <returns>
         /// Returns the circumference of the circle.
         /// </returns>
```

Fig. E.2 **`Circle`** class marked up with XML comments. (Part 3 of 4.)

```
      public double Circumference()
      {
         return Math.PI * Diameter();
      }

      /// <summary>
      /// Computes the area of the circle.
      /// </summary>
      /// <remarks>
      /// Uses constant <c><b>Math.PI</b></c>
      /// <see cref="System.Math.PI"/>
      /// </remarks>
      /// <returns>
      /// Returns the area of the circle.
      /// </returns>
      public double Area()
      {
         return Math.PI * Math.Pow( Radius, 2 );
      }

      /// <summary>
      /// Converts the <c><b>Circle</b></c> to
      /// <b>string</b> format.
      /// </summary>
      /// <remarks>
      /// Overrides <c><b>ToString</b></c> method of base class.
      /// <see cref="CircleTest.Point.ToString"/>
      /// </remarks>
      /// <returns>
      /// Returns a string that includes the center of the
      /// circle and its radius.
      /// </returns>
      public override string ToString()
      {
         return "Center = " + base.ToString() +
            "; Radius = " + Radius;
      }

   } // end class Circle
}
```

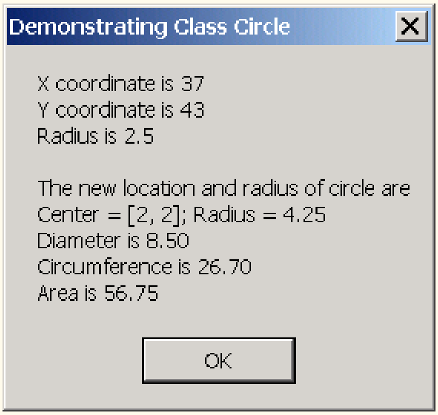
Fig. E.2 **`Circle`** class marked up with XML comments. (Part 4 of 4.)

```
// Fig. E.3: CircleTest.cs
// Manipulating a Circle object.

using System;
using System.Windows.Forms;

namespace CircleTest
{
```

Fig. E.3 **`CircleTest`** class marked up with XML comments. (Part 1 of 3.)

```
   /// <summary>
   /// Class <c><b>CircleTest</b></c> test the
   /// <c><b>Point</b></c> and <c><b>Point</b></c> classes.
   /// </summary>
   class CircleTest
   {
      /// <summary>
      /// Entry point of application.
      /// </summary>
      /// <remarks>
      /// In this application all command-line arguments
      /// are ignored.
      /// </remarks>
      /// <param name="args">
      /// Optional arguments to Main.
      /// </param>
      static void Main( string[] args )
      {
         Circle circle = new Circle( 37, 43, 2.5 );

         // append Circle properties to output
         string output = "X coordinate is " + circle.X + "\n" +
            "Y coordinate is " + circle.Y + "\n" +
            "Radius is " + circle.Radius;

         // set new coordinates and radius
         circle.X = 2;
         circle.Y = 2;
         circle.Radius = 4.25;

         output += "\n\n" +
            "The new location and radius of circle are " +
            "\n" + circle + "\n";

         // display Circle's Diameter
         output += "Diameter is " +
            String.Format( "{0:F}", circle.Diameter() ) + "\n";

         // display Circle's Circumference
         output += "Circumference is " +
           String.Format( "{0:F}", circle.Circumference() ) + "\n";

         // display Circle's Area
         output += "Area is " +
            String.Format( "{0:F}", circle.Area() );

         MessageBox.Show( output, "Demonstrating Class Circle" );

      } // end method Main

   } // end class CircleTest
}
```

Fig. E.3 `CircleTest` class marked up with XML comments. (Part 2 of 3.)



X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of circle are
Center = [2, 2]; Radius = 4.25
Diameter is 8.50
Circumference is 26.70
Area is 56.75

Fig. E.3 `CircleTest` class marked up with XML comments. (Part 3 of 3.)

XML documentation comments can be placed before a class definition, an interface definition, a constructor or a member (i.e., an instance variable or a reference). The programmer can place a description (i.e., purpose) of the class in element ***`summary`***. The **`summary`** element can contain as many lines as necessary to provide a description of the class method, properties, members, etc. As we will see in the next section, any content placed in element **`summary`** will be marked up in a column (labeled **Description**) of an HTML table. An example of a **`summary`** is shown on lines 8–11 of Fig. E.1 to provide a description of class **`Point`**. (We also used these tags in Section E.2 when we introduced documentation comments.)

Two elements commonly used to describe methods are ***`returns`*** and ***`param`***. Element **`returns`** contains information on the return value, as illustrated by lines 109–112 of Fig. E.1. Method **`ToString`** of **`Point`** returns a formatted **`string`** that has the point's *x-y* coordinate pair. Similarly, the ***`param`*** element contains information on a method's parameters. For example, lines 50–55 of Fig. E.1 associate one **`param`** element with variable **`x`**, and another **`param`** element with variable **`y`**.

We use **`c`** XML elements to mark up regions of code in our comments. Line 102 of Fig. E.2 shows the use of the **`c`** element to specify that **`Math.PI`** should be marked up as code in the resulting documentation. Notice that the **`c`** element contains **`b`** element that places **`Math.PI`** in boldface type on the Web page.

The ***`remarks`*** tag enables programmers to document any "miscellaneous" information or detailed comments. For example, lines 116–119 of Fig. E.2 documents that method **`Area`** uses the constant **`Math.PI`**.

The **`see`** tag (lines 103, 118 and 134 of Fig. E.2) is references another class or member (method, constant, property, etc.). Any member can be referenced by using the fully qualifying name (e.g., **`System.Console.ReadLine`**). The **`value`** tag (lines 67–70 and 88–91 of Fig. E.1 and lines 69–72 of Fig. E.2) is used to describe properties. These comments have no effect on the comment Web pages that can be generated.

For more information on these tags and other tags to use, visit the following URI:

```
ms-help://MS.VSCC/MS.MSDNVS/csref/html/
vclrftagsfordocumentationcomments.htm
```

E.4 Creating Comment Web Pages

In this section, we show how Visual Studio .NET can create documentation in Web-page format from source code that contains the XML documentation comments. We demonstrate this feature on the project containing the classes of Fig. E.1, Fig. E.2 and Fig. E.3. After opening this project, select **Tools > Build Comment Web Pages** (Fig. E.4). The **Build Comment Web Pages** window will appear, enabling the developer to specify the project(s) containing the files that Visual Studio .NET should document (Fig. E.5). If the developer selects **Build for entire Solution**, Visual Studio .NET will document all files in the current solution. If the developer selects **Build for selected Projects**, Visual Studio .NET will document only those files in the project that the developer specifies. In addition, the developer can specify the directory where Visual Studio .NET should store the generated HTML content. If the developer selects **Add to Favorites**, Visual Studio .NET will bookmark this content in the **Favorites** menu of Internet Explorer.

Press **OK** to generate the HTML content. Visual Studio immediately creates and displays the documentation using a style sheet. In our example, the user can view the communication from classes `Circle`, `CircleTest` and `Point` by selecting the desired class in the left-most column. Figure E.6 shows the documentation for class `Circle`.

Note that all member names and `summary` elements in Fig. E.2 have been formatted and placed in **Members** and **Description** columns, respectively (Fig. E.6). Selecting an item from the **Members** column opens an HTML page associated with that item. Figure E.7 shows the HTML page associated with method `Area` of class `Circle`. Notice that the `returns` tags in Fig. E.2 on lines 120–122 mark up the text that is documented as the text placed in the **Description** column.

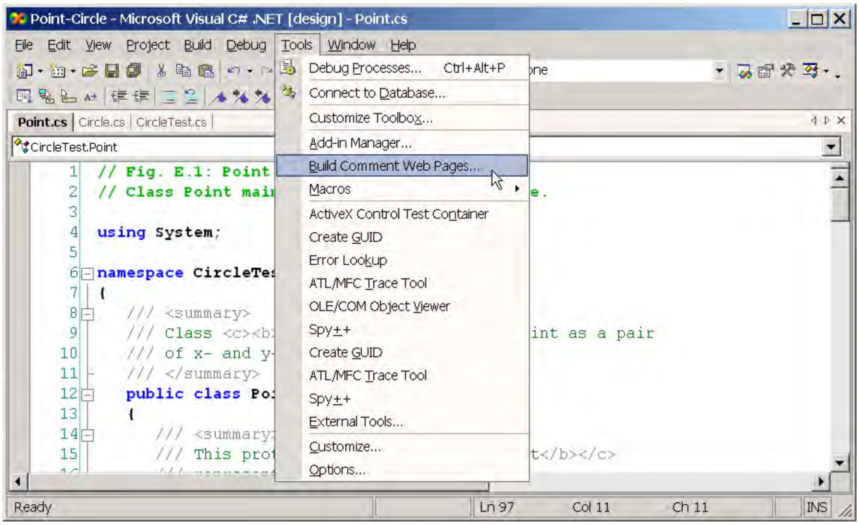
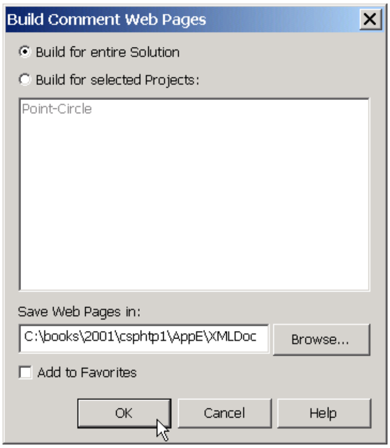

Point-Circle - Microsoft Visual C# .NET [design] - Point.cs
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Debug Processes... Ctrl+Alt+P
Connect to Database...
Customize Toolbox...
Add-in Manager...
Build Comment Web Pages...
Macros
ActiveX Control Test Container
Create GUID
Error Lookup
ATL/MFC Trace Tool
OLE/COM Object Viewer
Spy++
Create GUID
ATL/MFC Trace Tool
Spy++
External Tools...
Customize...
Options...
Point.cs
Circle.cs
CircleTest.cs
CircleTest.Point
// Fig. E.1: Point
// Class Point mai
e.
using System;
namespace CircleTes
{
/// <summary>
/// Class <c><b
int as a pair
/// of x- and y-
/// </summary>
public class Po
{
/// <summary
/// This prot
t</c>
Ready
Ln 97
Col 11
Ch 11
INS

Fig. E.4 Selecting the **Build Comment Web Pages** from **Tools** menu.


Build Comment Web Pages
OK
Cancel
Help
Browse...

● Build for entire Solution

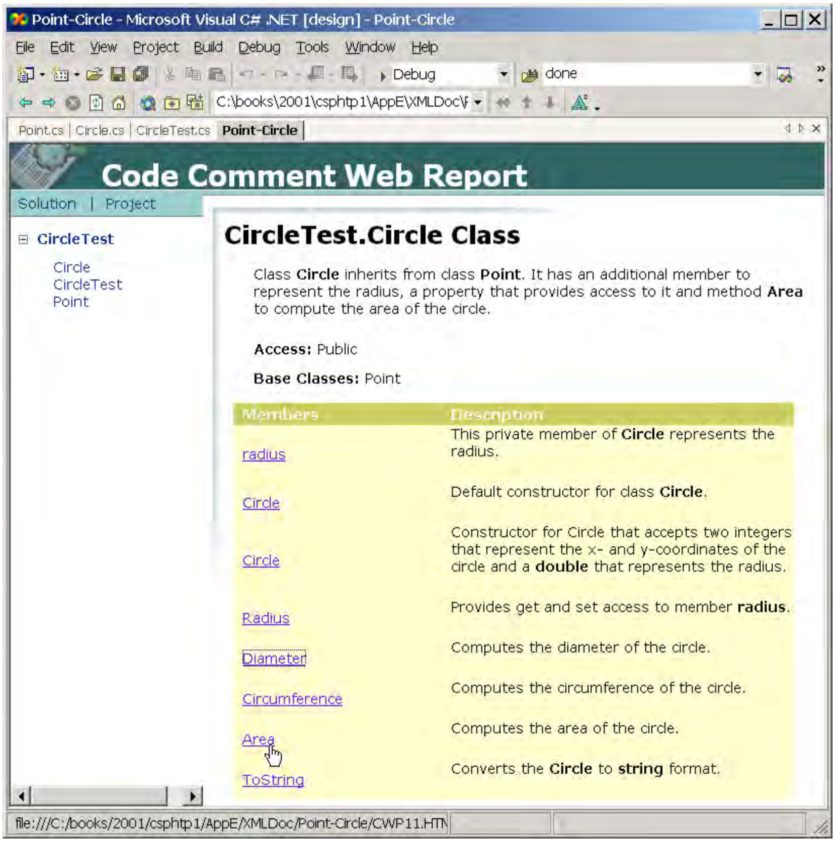
○ Build for selected Projects:

Point-Circle

Save Web Pages in: C:\books\2001\csphtp1\AppE\XMLDoc

☐ Add to Favorites

Fig. E.5 Saving a document to a file.


Point-Circle - Microsoft Visual C# .NET [design] - Point-Circle
File Edit View Project Build Debug Tools Window Help
Debug
done
C:\books\2001\csphtp1\AppE\XMLDoc\F
Point.cs
Circle.cs
CircleTest.cs
Point-Circle
Code Comment Web Report
Solution | Project
CircleTest
Circle
CircleTest
Point
file:///C:/books/2001/csphtp1/AppE/XMLDoc/Point-Circle/CWP11.HTM

CircleTest.Circle Class

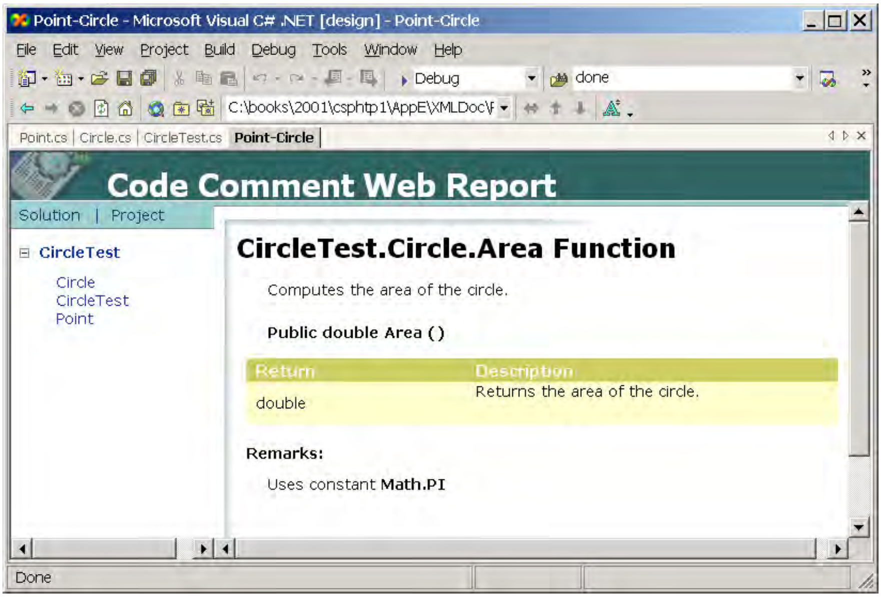
Class **Circle** inherits from class **Point**. It has an additional member to represent the radius, a property that provides access to it and method **Area** to compute the area of the circle.

Access: Public

Base Classes: Point

Members	Description
radius	This private member of **Circle** represents the radius.
Circle	Default constructor for class **Circle**.
Circle	Constructor for Circle that accepts two integers that represent the x- and y-coordinates of the circle and a **double** that represents the radius.
Radius	Provides get and set access to member **radius**.
Diameter	Computes the diameter of the circle.
Circumference	Computes the circumference of the circle.
Area	Computes the area of the circle.
ToString	Converts the **Circle** to **string** format.

Fig. E.6 XHTML documentation of class `Circle`.


Point-Circle - Microsoft Visual C# .NET [design] - Point-Circle
File Edit View Project Build Debug Tools Window Help
Debug
done
C:\books\2001\csphtp1\AppE\XMLDoc\F
Point.cs Circle.cs CircleTest.cs Point-Circle
Done

Code Comment Web Report

Solution | Project

CircleTest

Circle
CircleTest
Point

CircleTest.Circle.Area Function

Computes the area of the circle.

Public double Area ()

Return	Description
double	Returns the area of the circle.

Remarks:

Uses constant **Math.PI**

Fig. E.7 XHTML documentation of method **`Area`** method of class **`Circle`**.

E.5 Creating XML Documentation Files

In this section, we discuss how to generate an XML documentation file that contains all elements in the source code comments. An application then can read such a file and create custom documentation from its information.

To create an XML documentation file for a project, right-click on the project in the **Solution Explorer** and select **Properties**. Select the **Configuration** folder, then the **Build** tab. Change the **XML Documentation File** property to the name of the file that will store the XML documentation and click **OK**. If this file does not exist, Visual Studio will create the file and place it in the **`bin/Debug`** directory of the current project. Select **Build > Build Solution** to compile the project and create the XML document. Figure E.8 shows the XML document generated for the example in Fig. E.1–Fig. E.3.

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>Point-Circle</name>
   </assembly>
   <members>
```

Fig. E.8 XML documentation generated by Visual Studio .NET. (Part 1 of 6.)

```

        <member name="T:CircleTest.Circle">
           <summary>
              Class <c><b>Circle</b></c> inherits from class
              <c><b>Point</b></c>. It has an additional member to
              represent the radius, a property that provides
              accessto it and method <c><b>Area</b></c> to
              compute the area of the circle.
           </summary>
        </member>

        <member name="T:CircleTest.Point">
            <summary>
               Class <c><b>Point</b></c> defines a point as a pair
               of x and y coordinates.
            </summary>
        </member>

        <member name="F:CircleTest.Point.xCoordinate">
           <summary>
              This protected member of <c><b>Point</b></c>
              represents the x coordinate.
           </summary>
           <returns> The x coordinate as an integer.</returns>
        </member>

        <member name="F:CircleTest.Point.yCoordinate">
           <summary>
              This protected member of <c><b>Point</b></c>
              represents the x coordinate.
           </summary>
           <returns> The y coordinate as an integer.</returns>
        </member>

        <member name="M:CircleTest.Point.#ctor">
           <summary>
              Default constructor for class <c><b>Point</b></c>.
           </summary>
           <remarks>
            Sets properties <c><b>X</b></c> and <c><b>Y</b></c> to 0.
           </remarks>
        </member>

        <member name=
           "M:CircleTest.Point.#ctor(System.Int32,System.Int32)">
           <summary>
              Constructor for <c><b>Point</b></c>
              that accepts two integers
              that represent the x and y coordinates of the point.
           </summary>
           <remarks>
              Uses <c><b>X</b></c> and <c><b>Y</b></c>
              properties to set the coordinates of the point,
```

Fig. E.8 XML documentation generated by Visual Studio .NET. (Part 2 of 6.)

```
               <em>not</em> private members <c><b>x</b></c>
               and <c><b>y</b></c>.
            </remarks>
            <param name="xValue">
               The x coordinate of the circle
            </param>
            <param name="yValue">
               The y coordinate of the circle.
            </param>
         </member>

         <member name="M:CircleTest.Point.ToString">
            <summary>
               Converts the <c><b>Point</b></c> to
               <b>string</b> format.
            </summary>
            <returns>
               Returns a string in format:
               [x coordinate, y coordinate].
            </returns>
         </member>

         <member name="P:CircleTest.Point.X">
            <summary>
               Provides get and set access to member
               <c><b>x</b></c>.
            </summary>
            <value>
               <c><b>X</b></c> accesses the value of the
               <c><b>x</b></c> data member.
            </value>
         </member>

         <member name="P:CircleTest.Point.Y">
            <summary>
               Provides get and set access to member
               <c><b>y</b></c>.
            </summary>
            <value>
               <c><b>Y</b></c> accesses the value of the
               <c><b>y</b></c> data member.
            </value>
         </member>

         <member name="F:CircleTest.Circle.radius">
            <summary>
               This private member of <c><b>Circle</b></c>
               represents the radius.
            </summary>
         </member>

         <member name="M:CircleTest.Circle.#ctor">
            <summary>
```

Fig. E.8 XML documentation generated by Visual Studio .NET. (Part 3 of 6.)

```
            Default constructor for class <c><b>Circle</b></c>.
         </summary>
         <remarks>
            Sets the radius to 0.
         </remarks>
      </member>

      <member name="M:CircleTest.Circle.#ctor(System.Int32,
         System.Int32,System.Double)">
         <summary>
            Constructor for <c><b>Circle<b></c> that accepts two
            integersthat represent the x and y coordinates of the
            circle and a <b>double</b> that represents the radius.
         </summary>
         <remarks>
            Uses property <c><b>Radius</b></c> to set the radius
            of the circle, <em>not</em> private member
            <c><b>radius</b></c>.
         </remarks>
         <param name="xValue">
            The x coordinate of the circle
         </param>
         <param name="yValue">
            The y coordinate of the circle.
         </param>
         <param name="radiusValue">
             The radius of the circle.
         </param>
      </member>

      <member name="M:CircleTest.Circle.Diameter">
         <summary>
            Computes the diameter of the circle.
         </summary>
         <returns>
            Returns the diameter of the circle.
         </returns>
      </member>

      <member name="M:CircleTest.Circle.Circumerence">
         <summary>
            Computes the circumference of the circle.
         </summary>
         <remarks>
            Uses constant <c><b>Math.PI</b></c>
            <see cref="F:System.Math.PI"/>
         </remarks>
         <returns>
            Returns the circumference of the circle.
         </returns>
      </member>

      <member name="M:CircleTest.Circle.Area">
```

Fig. E.8 XML documentation generated by Visual Studio .NET. (Part 4 of 6.)

```
           <summary>
              Computes the area of the circle.
           </summary>
           <remarks>
              Uses constant <c><b>Math.PI</b></c>
              <see cref="F:System.Math.PI"/>
           </remarks>
           <returns>
              Returns the area of the circle.
           </returns>
        </member>

        <member name="M:CircleTest.Circle.ToString">
           <summary>
              Converts the <c><b>Circle</b></c> to
              <b>string</b> format.
           </summary>
           <remarks>
              Overrides <c><b>ToString</b></c> method of base class.
              <see cref="!:CircleTest.Point.ToString"/>
           </remarks>
           <returns>
              Returns a string that includes the center of the
              circle and its radius.
           </returns>
        </member>

        <member name="P:CircleTest.Circle.Radius">
           <summary>
              Provides get and set access to member
              <c><b>radius</b></c>.
           </summary>
           <remarks>
              The <c><b>set</b></c> method
              ensures that <c><b>radius</b></c>
              is <em>not</em> set to a
              negative number.
           </remarks>
           <value>
              <c><b>Radius</b></c> accesses the value of the
              <c><b>radius</b></c> data member.
           </value>
        </member>

        <member name="T:CircleTest.CircleTest">
           <summary>
              Class <c><b>CircleTest</b></c> inherits from class
              tests the <c><b>Point</b></c> and
              <c><b>Point</b></c> classes.
           </summary>
        </member>

        <member name="M:CircleTest.CircleTest.Main(System.String[])">
```

Fig. E.8 XML documentation generated by Visual Studio .NET. (Part 5 of 6.)

```
219          <summary>
220             Entry point of application.
221          </summary>
222          <remarks>
223             In this application all command-line arguments
224             are ignored.
225          </remarks>
226          <param name="args">
227             Optional arguments to Main.
228          </param>
229       </member>
230
231     </members>
232 </doc>
```

Fig. E.8 XML documentation generated by Visual Studio .NET. (Part 6 of 6.)

Notice that only class members are included in the generated XML file. Each class member has a ***`member`*** element which includes all XML comments for that member. For example, lines 50–69 define a `member` element that contains information on the two-argument `Point` constructor. The ***`name`*** attribute of a `member` tag is a `string` that contains information about the name and type of the member. The type is specified by a capital letter: `M` stands for method, `P` for property (or indexer), `E` for event and `T` for type (i.e, class). For a complete listing of these abbreviations, select **Help > Index**, then locate the topic **processing XML files in C#**. In Fig. E.8, line 51 contains the value of the `name` attribute and contains an `M` as the first letter, indicating that line 51 declares a method (recall that a constructor is a specialized method). A colon follows, after which the full name of the method is shown. For this example, it is `CircleTest.Point.#ctor(System.Int32,System.Int32)`. Because this is a constructor, the string `#ctor` is used in the fully qualified name. This constructor takes two `int` arguments—the parentheses after the name of each member specify that member's type.

TERMINOLOGY

`///` (documentation comment)
Build Comment Web Pages
`c` element
class definition
constructor
creating XML documentation
directory
documentation
Documentation column
HTML
instance variable
interface definition
member
`member` element
Members column
method declaration
`name` attribute of `member` element
`para` element
`param` element
parameters
property
reference
`remarks` element
return value
`returns` element
`see` element
source code
style sheet
`summary` element
tag
`value` element
XML documentation comment

SUMMARY

- Programmers should document specific information on a class, such as the class's role in a system, the functionality that each method provides for the class and the purpose of each class variable.
- Documentation helps all programmers understand how classes can interoperate, as well as facilitate modification, use and extension of each class.
- Visual Studio .NET provides the XML documentation tool. This tool converts key pieces of information in the code—such as the class's members, the hierarchy to which the class belongs and any other general remarks the programmer wishes to document—to HTML or XML format.
- The programmer specifies the general remarks to be documented by placing them in special regions in the code, called XML documentation comments.
- The documentation-generation tool recognizes only single-line comments that begin with three forward slashes (`///`).
- The compiler does not translate documentation comments to MSIL (Microsoft Intermediate Language).
- The programmer can place a description (i.e., purpose) of the class in between **`summary`** tags.
- Element **`returns`** contains information on the return value. Similarly, the **`param`** element contains information on a method's parameters.
- Element **`c`** marked up regions of code in the comments.
- The **`remarks`** tag enables programmers to document any "miscellaneous" information or detailed comments relating to a member.
- The **`see`** tag is used to reference another member (method, constant, property, etc.).

F

ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. F.1 ASCII character set.

The digits at the left of the table are the left digits of the decimal equivalent (0–127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for “F” is 70, and the character code for “&” is 38.

Most users of this book are interested in the ASCII character set used to represent English characters on many computers. The ASCII character set is a subset of the Unicode character set used by C# to represent characters from most of the world’s languages. For more information on the Unicode character set, see Appendix G.

G

Unicode®

Objectives

- To become familiar with Unicode.
- To discuss the mission of the Unicode Consortium.
- To discuss the design basis of Unicode.
- To understand the three Unicode encoding forms: UTF-8, UTF-16 and UTF-32.
- To introduce characters and glyphs.
- To discuss the advantages and disadvantages of using Unicode.
- To provide a brief tour of the Unicode Consortium's Web site.

Outline

G.1 Introduction
G.2 Unicode Transformation Formats
G.3 Characters and Glyphs
G.4 Advantages and Disadvantages of Unicode
G.5 Unicode Consortium's Web Site
G.6 Using Unicode
G.7 Character Ranges

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

G.1 Introduction

The use of inconsistent character *encodings* (i.e., numeric values associated with characters) when developing global software products causes serious problems because computers process information using numbers. For example, the character "a" is converted to a numeric value so that a computer can manipulate that piece of data. Many countries and corporations have developed their own encoding systems that are incompatible with the encoding systems of other countries and corporations. For example, the Microsoft Windows operating system assigns the value 0xC0 to the character "A with a grave accent," while the Apple Macintosh operating system assigns that same value to an upside-down question mark. This results in the misrepresentation and possible corruption of data because the data is not processed as intended.

In the absence of a widely implemented universal character encoding standard, global software developers had to *localize* their products extensively before distribution. Localization includes the language translation and cultural adaptation of content. The process of localization usually includes significant modifications to the source code (such as the conversion of numeric values and the underlying assumptions made by programmers), which results in increased costs and delays releasing the software. For example, some English-speaking programmers might design global software products assuming that a single character can be represented by one byte. However, when those products are localized for Asian markets, the programmer's assumptions are no longer valid; thus, the majority, if not the entirety, of the code needs to be rewritten. Localization is necessary with each release of a version. By the time a software product is localized for a particular market, a newer version, which needs to be localized as well, may be ready for distribution. As a result, it is cumbersome and costly to produce and distribute global software products in a market where there is no universal character encoding standard.

In response to this situation, the *Unicode Standard*, an encoding standard that facilitates the production and distribution of software, was created. The Unicode Standard outlines a specification to produce consistent encoding of the world's characters and *symbols*. Software products that handle text encoded in the Unicode Standard need to be localized, but the localization process is simpler and more efficient because the numeric values need not be converted and the assumptions made by programmers about the character encoding are universal. The Unicode Standard is maintained by a nonprofit organization called the

Unicode Consortium, whose members include Apple, IBM, Microsoft, Oracle, Sun Microsystems, Sybase and many others.

When the Consortium envisioned and developed the Unicode Standard, they wanted an encoding system that was *universal*, *efficient*, *uniform* and *unambiguous*. A universal encoding system encompasses all commonly used characters. An efficient encoding system allows text files to be parsed easily. A uniform encoding system assigns fixed values to all characters. An unambiguous encoding system represents a given character in a consistent manner. These four terms are referred to as the Unicode Standard *design basis*.

G.2 Unicode Transformation Formats

Although Unicode incorporates the limited ASCII *character set* (i.e., a collection of characters), it encompasses a more comprehensive character set. In ASCII, each character is represented by a byte containing 0s and 1s. One byte is capable of storing the binary numbers from 0 to 255. Each character is assigned a number between 0 and 255; thus, ASCII-based systems can support only 256 characters, a tiny fraction of the world's characters. Unicode extends the ASCII character set by encoding the vast majority of the world's characters. The Unicode Standard encodes all of those characters in a uniform numerical space from 0 to 10FFFF hexadecimal. An implementation will express these numbers in one of several transformation formats, choosing the one that best fits the particular application at hand.

Three such formats are in use, called *UTF-8*, *UTF-16* and *UTF-32*, depending on the size of the units—in *bits*—being used. UTF-8, a variable width encoding form, requires one to four bytes to express each Unicode character. UTF-8 data consists of 8-bit bytes (sequences of one, two, three or four bytes depending on the character being encoded) and is well suited for ASCII-based systems when there is a predominance of one-byte characters (ASCII represents characters as one-byte). Currently, UTF-8 is widely implemented in UNIX systems and in databases. [*Note*: Currently, Internet Explorer 5.5 and Netscape Communicator 6 only support UTF-8, so document authors should use UTF-8 for encoding XML and XHTML documents.]

The variable width UTF-16 encoding form expresses Unicode characters in units of 16-bits (i.e., as two adjacent bytes, or a short integer in many machines). Most characters of Unicode are expressed in a single 16-bit unit. However, characters with values above FFFF hexadecimal are expressed with an ordered pair of 16-bit units called *surrogates*. Surrogates are 16-bit integers in the range D800 through DFFF, which are used solely for the purpose of "escaping" into higher numbered characters. Approximately one million characters can be expressed in this manner. Although a surrogate pair requires 32 bits to represent characters, it is space efficient to use these 16-bit units. Surrogates are rare characters in current implementations. Many string-handling implementations are written in terms of UTF-16. [*Note*: Details and sample code for UTF-16 handling are available on the Unicode Consortium Web site at **`www.unicode.org`**.]

Implementations that require significant use of rare characters or entire scripts encoded above FFFF hexadecimal should use UTF-32, a 32-bit, fixed-width encoding form that usually requires twice as much memory as UTF-16 encoded characters. The major advantage of the fixed-width UTF-32 encoding form is that it expresses all characters uniformly, so it is easy to handle in arrays.

There are few guidelines that state when to use a particular encoding form. The best encoding form to use depends on computer systems and business protocols, not on the data.

Typically, the UTF-8 encoding form should be used where computer systems and business protocols require data to be handled in 8-bit units, particularly in legacy systems being upgraded because it often simplifies changes to existing programs. For this reason, UTF-8 has become the encoding form of choice on the Internet. Likewise, UTF-16 is the encoding form of choice on Microsoft Windows applications. UTF-32 is likely to become more widely used in the future as more characters are encoded with values above FFFF hexadecimal. Also, UTF-32 requires less sophisticated handling than UTF-16 in the presence of surrogate pairs. Figure G.1 shows the different ways in which the three encoding forms handle character encoding.

G.3 Characters and Glyphs

The Unicode Standard consists of *characters*, written components (i.e., alphabetic letters, numerals, punctuation marks, accent marks, etc.) that can be represented by numeric values. Examples of characters include U+0041 LATIN CAPITAL LETTER A. In the first character representation, U+*yyyy* is a *code value*, in which U+ refers to Unicode code values, as opposed to other hexadecimal values. The *yyyy* represents a four-digit hexadecimal number of an encoded character. Code values are bit combinations that represent encoded characters. Characters are represented using *glyphs*, various shapes, fonts and sizes for displaying characters. There are no code values for glyphs in the Unicode Standard. Examples of glyphs are shown in Fig. G.2.

The Unicode Standard encompasses the alphabets, ideographs, syllabaries, punctuation marks, *diacritics*, mathematical operators, etc. that comprise the written languages and scripts of the world. A diacritic is a special mark added to a character to distinguish it from another letter or to indicate an accent (e.g., in Spanish, the tilde "~" above the character "n"). Currently, Unicode provides code values for 94,140 character representations, with more than 880,000 code values reserved for future expansion.

Character	**UTF-8**	**UTF-16**	**UTF-32**
LATIN CAPITAL LETTER A	0x41	0x0041	0x00000041
GREEK CAPITAL LETTER ALPHA	0xCD 0x91	0x0391	0x00000391
CJK UNIFIED IDEOGRAPH-4E95	0xE4 0xBA 0x95	0x4E95	0x00004E95
OLD ITALIC LETTER A	0xF0 0x80 0x83 0x80	0xDC00 0xDF00	0x00010300

Fig. G.1 Correlation between the three encoding forms.

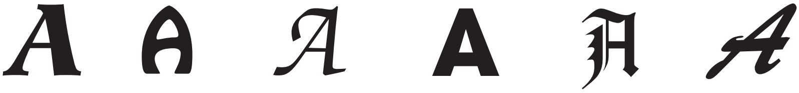


Fig. G.2 Various glyphs of the character A.

G.4 Advantages and Disadvantages of Unicode

The Unicode Standard has several significant advantages that promote its use. One is the impact it has on the performance of the international economy. Unicode standardizes the characters for the world's writing systems to a uniform model that promotes transferring and sharing data. Programs developed using such a schema maintain their accuracy because each character has a single definition (i.e., *a* is always U+0061, *%* is always U+0025). This enables corporations to manage the high demands of international markets by processing different writing systems at the same time. Also, all characters can be managed in an identical manner, thus avoiding any confusion caused by different character code architectures. Moreover, managing data in a consistent manner eliminates data corruption, because data can be sorted, searched and manipulated using a consistent process.

Another advantage of the Unicode Standard is *portability* (i.e., the ability to execute software on disparate computers or with disparate operating systems). Most operating systems, databases, programming languages and Web browsers currently support, or are planning to support, Unicode. Additionally, Unicode includes more characters than any other character set in common use (although it does not yet include all of the world's characters).

A disadvantage of the Unicode Standard is the amount of memory required by UTF-16 and UTF-32. ASCII character sets are 8 bits in length, so they require less storage than the default 16-bit Unicode character set. However, the *double-byte character set (DBCS)* and the *multi-byte character set (MBCS)* that encode Asian characters (ideographs) require two to four bytes, respectively. In such instances, the UTF-16 or the UTF-32 encoding forms may be used with little hindrance on memory and performance.

G.5 Unicode Consortium's Web Site

If you would like to learn more about the Unicode Standard, visit **`www.unicode.org`**. This site provides a wealth of information about the Unicode Standard. Currently, the home page is organized into various sections: *New to Unicode*, *General Information*, *The Consortium*, *The Unicode Standard*, *Work in Progress* and *For Members*.

The *New to Unicode* section consists of two subsections: ***What is Unicode?*** and ***How to Use this Site***. The first subsection provides a technical introduction to Unicode by describing design principles, character interpretations and assignments, text processing and Unicode conformance. This subsection is recommended reading for anyone new to Unicode. Also, this subsection provides a list of related links that provide the reader with additional information about Unicode. The **How to Use this Site** subsection contains information about using and navigating the site as well hyperlinks to additional resources.

The *General Information* section contains six subsections: ***Where is my Character?***, ***Display Problems?***, ***Useful Resources***, ***Enabled Products***, ***Mail Lists*** and ***Conferences***. The main areas covered in this section include a link to the Unicode code charts (a complete listing of code values) assembled by the Unicode Consortium as well as a detailed outline on how to locate an encoded character in the code chart. Also, the section contains advice on how to configure different operating systems and Web browsers so that the Unicode characters can be viewed properly. Moreover, from this section, the user can navigate to other sites that provide information on various topics such as, fonts, linguistics and other standards such as the *Armenian Standards Page* and the *Chinese GB 18030 Encoding Standard*.

The Consortium section consists of five subsections: ***Who we are***, ***Our Members***, ***How to Join***, ***Press Info*** and ***Contact Us***. This section provides a list of the current Unicode Consortium members as well as information on how to become a member. Privileges for each member type—*full*, *associate*, *specialist* and *individual*—and the fees assessed to each member are listed here.

The Unicode Standard section consists of nine subsections: ***Start Here***, ***Latest Version***, ***Technical Reports***, ***Code Charts***, ***Unicode Data***, ***Updates & Errata***, ***Unicode Policies***, ***Glossary*** and ***Technical FAQ***. This section describes the updates applied to the latest version of the Unicode Standard, as well as categorizing all defined encoding. The user can learn how the latest version has been modified to encompass more features and capabilities. For instance, one enhancement of Version 3.1 is that it contains additional encoded characters. Also, if users are unfamiliar with vocabulary terms used by the Unicode Consortium, then they can navigate to the **Glossary** subsection.

The *Work in Progress* section consists of three subsections: ***Calendar of Meetings***, ***Proposed Characters*** and ***Submitting Proposals***. This section presents the user with a catalog of the recent characters included into the Unicode Standard scheme as well as those characters being considered for inclusion. If users determine that a character has been overlooked, then they can submit a written proposal for the inclusion of that character. The **Submitting Proposals** subsection contains strict guidelines that must be adhered to when submitting written proposals.

The *For Members* section consists of two subsections: ***Member Resources*** and ***Working Documents***. These subsections are password protected; only consortium members can access these links.

G.6 Using Unicode

Visual Studio .NET uses Unicode UTF-16 encoding to represent all characters. Figure G.3 uses C# to display the text “Welcome to Unicode!” in eight different languages: English, French, German, Japanese, Portuguese, Russian, Spanish and Simplified Chinese. [*Note*: The Unicode Consortium’s Web site contains a link to code charts that lists the 16-bit Unicode code values.]

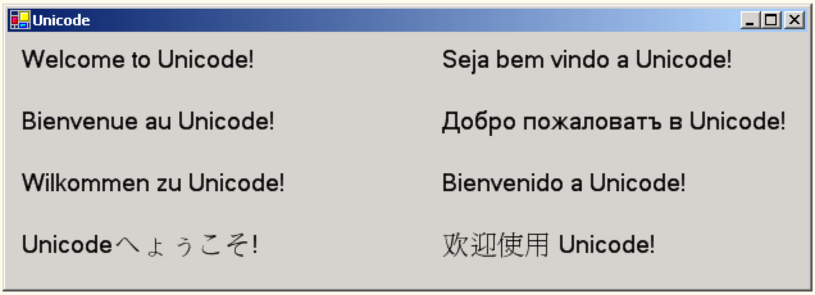
```
// Fig F.3: Unicode.cs
// Using unicode encoding

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class Unicode : System.Windows.Forms.Form
{
   internal System.Windows.Forms.Label lblChinese;
   internal System.Windows.Forms.Label lblSpanish;
   internal System.Windows.Forms.Label lblRussian;
```

Fig. G.3 Unicode values for multiple languages. (Part 1 of 3.)

```
   internal System.Windows.Forms.Label lblPortuguese;
   internal System.Windows.Forms.Label lblJapanese;
   internal System.Windows.Forms.Label lblGerman;
   internal System.Windows.Forms.Label lblFrench;
   internal System.Windows.Forms.Label lblEnglish;
   private System.ComponentModel.Container components = null;

   // Visual Studio .NET generated code

   // main entry point for the application.
   [STAThread]
   static void Main()
   {
      Application.Run(new Unicode());
   }

   private void Unicode_Load(object sender, System.EventArgs e)
   {
      // English
      char[] english = {'\u0057', '\u0065', '\u006C',
         '\u0063', '\u006F', '\u006D', '\u0065', '\u0020',
         '\u0074', '\u006F', '\u0020' };

      lblEnglish.Text = new string(english) +
         "Unicode" + '\u0021';

      // French
      char[] french = { '\u0042', '\u0069', '\u0065',
         '\u006E', '\u0076', '\u0065', '\u006E', '\u0075',
         '\u0065', '\u0020', '\u0061', '\u0075', '\u0020' };

      lblFrench.Text = new string(french) +
         "Unicode" + '\u0021';

      // German
      char[] german = {'\u0057', '\u0069', '\u006C',
         '\u006B', '\u006F', '\u006D', '\u006D', '\u0065',
         '\u006E', '\u0020', '\u007A', '\u0075', '\u0020'};

      lblGerman.Text =  new string(german) +
         "Unicode" + '\u0021';

      // Japanese
      char[] japanese = { '\u3078',  '\u3087', '\u3045',
         '\u3053', '\u305D', '\u0021'};

      lblJapanese.Text = "Unicode" + new string(japanese);

      // Portuguese
      char[] portuguese = {'\u0053', '\u0065', '\u006A',
         '\u0061', '\u0020', '\u0062', '\u0065', '\u006D',
         '\u0020', '\u0076', '\u0069', '\u006E', '\u0064',
         '\u006F', '\u0020', '\u0061', '\u0020' };
```

Fig. G.3 Unicode values for multiple languages. (Part 2 of 3.)

```
69
70       lblPortuguese.Text = new string(portuguese) +
71          "Unicode" + '\u0021';
72
73       // Russian
74      char[] russian = {  '\u0414', '\u043E', '\u0431',
75         '\u0440', '\u043E', '\u0020', '\u043F', '\u043E',
76         '\u0436', '\u0430', '\u043B', '\u043E', '\u0432',
77         '\u0430', '\u0442', '\u044A', '\u0020', '\u0432',
78         '\u0020' };
79
80       lblRussian.Text = new string(russian) +
81          "Unicode" + '\u0021';
82
83       // Spanish
84       char[] spanish = {'\u0042', '\u0069', '\u0065',
85          '\u006E', '\u0076', '\u0065', '\u006E', '\u0069',
86          '\u0064', '\u006F', '\u0020', '\u0061', '\u0020' };
87
88       lblSpanish.Text = new string(spanish) +
89          "Unicode" + '\u0021';
90
91       // Simplified Chinese
92       char[] chinese = {'\u6B22', '\u8FCE', '\u4F7F',
93          '\u7528', '\u0020' };
94
95       lblChinese.Text = new string(chinese) +
96          "Unicode" + '\u0021';
97    }  // end method Unicode_Load
98 } // end class Unicode
```



Welcome to Unicode!

Seja bem vindo a Unicode!

Bienvenue au Unicode!

Добро пожаловатъ в Unicode!

Wilkommen zu Unicode!

Bienvenido a Unicode!

Unicodeへょぅこそ!

欢迎使用 Unicode!

Fig. G.3 Unicode values for multiple languages. (Part 3 of 3.)

Lines 35–37 contain the hexadecimal codes for the English text. The **Code Charts** page on the Unicode Consortium Web site contains a document that lists the code values for the **Basic Latin** *block* (or category), which includes the English alphabet. The hexadecimal codes in lines 35–36 equate to "**Welcome** ". When using Unicode characters in C#, the format `\u`*yyyy* is used, where *yyyy* represents the hexadecimal Unicode encoding. For example, the letter "W" (in "Welcome") is denoted by `\u0057`. Line 36 contains the hexadecimal for the *space* character (`\u0020`). The unicode value for the word "**to** " is on line 37. Lines 39–40 create a new string from the character array and append the word "Unicode.” “Unicode” is not encoded because it is a registered trademark and has no equivalent translation in most languages. Line 40 also contains the `\u0021` notation for the exclamation mark (`!`).

The remaining welcome messages (lines 43–96) contain the unicode values for the other seven languages. The code values used for the French, German, Portuguese and Spanish text are located in the **Basic Latin** block, the code values used for the Simplified Chinese text are located in the **CJK Unified Ideographs** block, the code values used for the Russian text are located in the **Cyrillic** block and the code values used for the Japanese text are located in the **Hiragana** block.

[*Note*: To render the Asian characters in a Windows application, you may need to install the proper language files on your computer. To do this in Windows 2000, open the **Regional Options** dialog from the **Control Panel** (**Start > Settings > Control Panel**). At the bottom of the **General** tab is a list of languages. Check the **Japanese** and the **Traditional Chinese** checkboxes and press **Apply**. Follow the directions of the install wizard to install the languages. For additional assistance, visit **`www.unicode.org/help/display_problems.html`**.]

G.7 Character Ranges

The Unicode Standard assigns code values, which range from `0000` (**Basic Latin**) to `E007F` (***Tags***), to the written characters of the world. Currently, there are code values for 94,140 characters. To simplify the search for a character and its associated code value, the Unicode Standard generally groups code values by *script* and function (i.e., Latin characters are grouped in a block, mathematical operators are grouped in another block, etc.). As a rule, a script is a single writing system that is used for multiple languages (e.g., the Latin script is used for English, French, Spanish, etc.) The **Code Charts** page on the Unicode Consortium Web site lists all the defined blocks and their respective code values. Figure G.4 lists some blocks (scripts) from the Web site and their range of code values.

Script	Range of Code Values
Arabic	`U+0600–U+06FF`
Basic Latin	`U+0000–U+007F`
Bengali (India)	`U+0980–U+09FF`
Cherokee (Native America)	`U+13A0–U+13FF`
CJK Unified Ideographs (East Asia)	`U+4E00–U+9FAF`
Cyrillic (Russia and Eastern Europe)	`U+0400–U+04FF`
Ethiopic	`U+1200–U+137F`
Greek	`U+0370–U+03FF`
Hangul Jamo (Korea)	`U+1100–U+11FF`
Hebrew	`U+0590–U+05FF`
Hiragana (Japan)	`U+3040–U+309F`

Fig. G.4 Some character ranges. (Part 1 of 2.)

Script	Range of Code Values
Khmer (Cambodia)	`U+1780–U+17FF`
Lao (Laos)	`U+0E80–U+0EFF`
Mongolian	`U+1800–U+18AF`
Myanmar	`U+1000–U+109F`
Ogham (Ireland)	`U+1680–U+169F`
Runic (Germany and Scandinavia)	`U+16A0–U+16FF`
Sinhala (Sri Lanka)	`U+0D80–U+0DFF`
Telugu (India)	`U+0C00–U+0C7F`
Thai	`U+0E00–U+0E7F`

Fig. G.4 Some character ranges. (Part 2 of 2.)

SUMMARY

- Before Unicode, software developers were plagued by the use of inconsistent character encoding (i.e., numeric values for characters). Most countries and organizations had their own encoding systems, which were incompatible. A good example is the individual encoding systems on the Windows and Macintosh platforms. Computers process data by converting characters to numeric values. For instance, the character “a” is converted to a numeric value so that a computer can manipulate that piece of data.
- Without Unicode, localization of global software requires significant modifications to the source code, which results in increased cost and in delays releasing the product.
- Localization is necessary with each release of a version. By the time a software product is localized for a particular market, a newer version, which needs to be localized as well, is ready for distribution. As a result, it is cumbersome and costly to produce and distribute global software products in a market where there is no universal character encoding standard.
- The Unicode Consortium developed the Unicode Standard in response to the serious problems created by multiple character encodings and the use of those encodings.
- The Unicode Standard facilitates the production and distribution of localized software. It outlines a specification for the consistent encoding of the world’s characters and symbols.
- Software products which handle text encoded in the Unicode Standard need to be localized, but the localization process is simpler and more efficient because the numeric values need not be converted.
- The Unicode Standard is designed to be universal, efficient, uniform and unambiguous.
- A universal encoding system encompasses all commonly used characters; an efficient encoding system parses text files easily; a uniform encoding system assigns fixed values to all characters; and an unambiguous encoding system represents the same character for any given value.
- Unicode extends the limited ASCII character set to include all the major characters of the world.
- Unicode makes use of three Unicode Transformation Formats (UTF): UTF-8, UTF-16 and UTF-32, each of which may be appropriate for use in different contexts.
- UTF-8 data consists of 8-bit bytes (sequences of one, two, three or four bytes depending on the character being encoded) and is well suited for ASCII-based systems when there is a predominance of one-byte characters (ASCII represents characters as one-byte).

- UTF-8 is a variable-width encoding form that is more compact for text involving mostly Latin characters and ASCII punctuation.
- UTF-16 is the default encoding form of the Unicode Standard. It is a variable width encoding form that uses 16-bit code units instead of bytes. Most characters are represented by a single unit, but some characters require surrogate pairs.
- Surrogates are 16-bit integers in the range D800 through DFFF, which are used solely for the purpose of "escaping" into higher numbered characters.
- Without surrogate pairs, the UTF-16 encoding form can only encompass 65,000 characters, but with the surrogate pairs, this is expanded to include over a million characters.
- UTF-32 is a 32-bit encoding form. The major advantage of the fixed-width encoding form is that it uniformly expresses all characters, so that they are easy to handle in arrays and so forth.
- The Unicode Standard consists of characters. A character is any written component that can be represented by a numeric value.
- Characters are represented using glyphs, various shapes, fonts and sizes for displaying characters.
- Code values are bit combinations that represent encoded characters. The Unicode notation for a code value is U+*yyyy* in which U+ refers to the Unicode code values, as opposed to other hexadecimal values. The *yyyy* represents a four-digit hexadecimal number.
- Currently, the Unicode Standard provides code values for 94,140 character representations.
- An advantage of the Unicode Standard is its impact on the overall performance of the international economy. Applications that conform to an encoding standard can be processed easily by computers anywhere.
- Another advantage of the Unicode Standard is its portability. Applications written in Unicode can be easily transferred to different operating systems, databases, Web browsers, etc. Most companies currently support, or are planning to support, Unicode.
- To obtain more information about the Unicode Standard and the Unicode Consortium, visit **`www.unicode.org`**. It contains a link to the code charts, which contain the 16-bit code values for the currently encoded characters.
- The Unicode Standard has become the default encoding system for XML and any language derived from XML, such as XHTML.
- The C# IDE uses Unicode UTF-16 encoding to represent all characters.
- When marking up C# documents, the entity reference **`\u`***yyyy* is used, where *yyyy* represents the hexadecimal code value.

TERMINOLOGY

`\u`*yyyy* notation
ASCII
block
character
character set
code value
diacritic
double-byte character set (DBCS)
efficient (Unicode design basis)
encode
entity reference
glyph
hexadecimal notation
localization
multi-byte character set (MBCS)
portability
script
surrogate
symbol
unambiguous (Unicode design basis)
Unicode Consortium
Unicode design basis
Unicode Standard
Unicode Transformation Format (UTF)

uniform (Unicode design basis)
universal (Unicode design basis)
UTF-16
UTF-32
UTF-8

SELF-REVIEW EXERCISES

G.1 Fill in the blanks in each of the following statements.

a) Global software developers had to __________ their products to a specific market before distribution.
b) The Unicode Standard is an __________ standard that facilitates the uniform production and distribution of software products.
c) The four design basis that constitute the Unicode Standard are: __________, __________, __________ and __________.
d) A __________ is the smallest written component the can be represented with a numeric value.
e) Software that can execute on different operating systems is said to be __________.
f) Of the three encoding forms, __________ is currently supported by Internet Explorer 5.5 and Netscape Communicator 6.

G.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The Unicode Standard encompasses all the world's characters.
b) A Unicode code value is represented as U+*yyyy*, where *yyyy* represents a number in binary notation.
c) A diacritic is a character with a special mark that emphasizes an accent.
d) Unicode is portable.
e) When designing C# programs, the entity reference is denoted by `#U+`*yyyy*.

ANSWERS TO SELF-REVIEW EXERCISES

G.1 a) localize. b) encoding. c) universal, efficient, uniform, unambiguous. d) character. e) portable. f) UTF-8.

G.2 a) False. It encompasses the majority of the world's characters. b) False. The *yyyy* represents a hexadecimal number. c) False. A diacritic is a special mark added to a character to distinguish it from another letter or to indicate an accent. d) True. e) False. The entity reference is denoted by `\u`*yyyy*.

EXERCISES

G.3 Navigate to the Unicode Consortium Web site (`www.unicode.org`) and write the hexadecimal code values for the following characters. In which block are they located?

a) Latin letter 'Z.'
b) Latin letter 'n' with the 'tilde (~).'
c) Greek letter 'delta.'
d) Mathematical operator 'less than or equal to.'
e) Punctuation symbol 'open quote (").'

G.4 Describe the Unicode Standard design basis.

G.5 Define the following terms:

a) code value.
b) surrogates.
c) Unicode Standard.
d) UTF-8.

e) UTF-16.
f) UTF-32.

G.6 Describe a scenario where it is optimal to store your data in UTF-16 format.

G.7 Using the Unicode Standard code values, create a C# document that prints your first and last name. If you know other writing systems, print your first and last name in those as well. Use a Windows form to render the document.

G.8 Write an ASP.NET program that prints "Welcome to Unicode!" in English, French, German, Japanese, Portuguese, Russian, Spanish and Traditional Chinese. Use the code values provided in Fig. G.3. In ASP.NET, a code value is represented the same way as in a Windows application (`\u`*yyyy*, where *yyyy* is a four-digit hexadecimal number).

H

COM Integration

H.1 Introduction

Initially, applications created for Windows or DOS were designed as single *monolithic executables*—i.e., complete applications packaged as single executable files. However, as software became more complex, developers began to experience difficulties constructing all the necessary components of an application. Furthermore, as the size of applications increased, it became impractical to redistribute an entire application to accommodate each application upgrade or bug fix.

To address these problems, Microsoft incorporated *shared libraries* into Windows, enabling developers to reuse and modularize code. A shared library, or *dynamic link library* (*DLL*) in Windows, is a file that contains compiled code that an application loads at execution time. The fact that these libraries are loaded at runtime allows developers to modify specific libraries and test the results without rebuilding an entire application. Multiple applications can use a single shared library, which reduces the overall memory requirements for running those applications. The partitioning of programs into small pieces also makes it easier to distribute application upgrades, because only the modified DLLs must be redistributed.

The introduction of shared libraries solved many problems that previously had restricted modularity and code reusability. However, the libraries also raised new concerns. Monolithic applications rarely created version conflicts—if an application vendor fixed a bug in one piece of software, it was unlikely that the upgrade would affect any other software on the system. With the establishment of system-wide shared libraries, a vendor's upgrade or modification of a library could "break" software that used an older version of that library. Often, developers packaged DLLs with their applications to ensure software compatibility. However, the packaged DLLs could overwrite preexisting libraries on users' systems, possibly affecting previously installed software. Problems introduced by shared libraries were so difficult to locate and fix that their effects became known as "DLL hell."

Microsoft developed the *Component Object Model (COM)* in an attempt to expand DLL functionality and correct DLL problems. COM is a specification that controls library versions and backwards compatibility and defines a communication standard among libraries. Microsoft defined the COM specification to be detailed and strict, thus ensuring that COM developers create compatible libraries. Microsoft also implemented the COM architecture on a large scale—virtually all Windows libraries adhere to the COM specification.

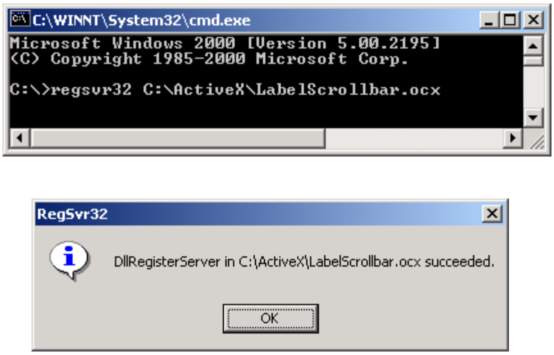
When implemented correctly, COM ensures highly organized and reusable libraries, but the specification does have limitations. For example, COM is difficult to program and deploy, because developers must guarantee that new COM components are both compatible with previous versions and correctly registered with the system. If a COM library is placed on a system without proper registration, the component will not be represented correctly in the registry, and applications might be unable to find or use the library.

In the .NET platform, COM components are no longer necessary. Microsoft .NET components retain the benefits of COM while resolving many of its associated problems. Components in .NET maintain all identification information internally—the independence of the component from the Windows Registry ensures correct component identification. Many developers and companies have invested significant resources in the development of COM components, but want to harness the power, organization and functionality of .NET. To facilitate developers' migration from COM to .NET, Microsoft created a set of tools designed to integrate existing COM components into the .NET platform.

H.2 ActiveX Integration

In recent years, ActiveX controls have been popular COM components. C# allows developers to import and use ActiveX controls in Windows Forms applications. We include an ActiveX `LabelScrollbar` control on the CD that accompanies this book which we now use to demonstrate Microsoft's ActiveX integration utility.

To use this control, students first must register the `.OCX` file in the Windows Registry. To register the control, open a **Command Prompt** and invoke the `RegSvr32` utility program which is located in the directory `c:\winnt\system32`. (This path might vary among different computers and Windows versions.) Figure H.1 depicts the successful registration of the `LabelScrollbar` ActiveX control.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>regsvr32 C:\ActiveX\LabelScrollbar.ocx
```

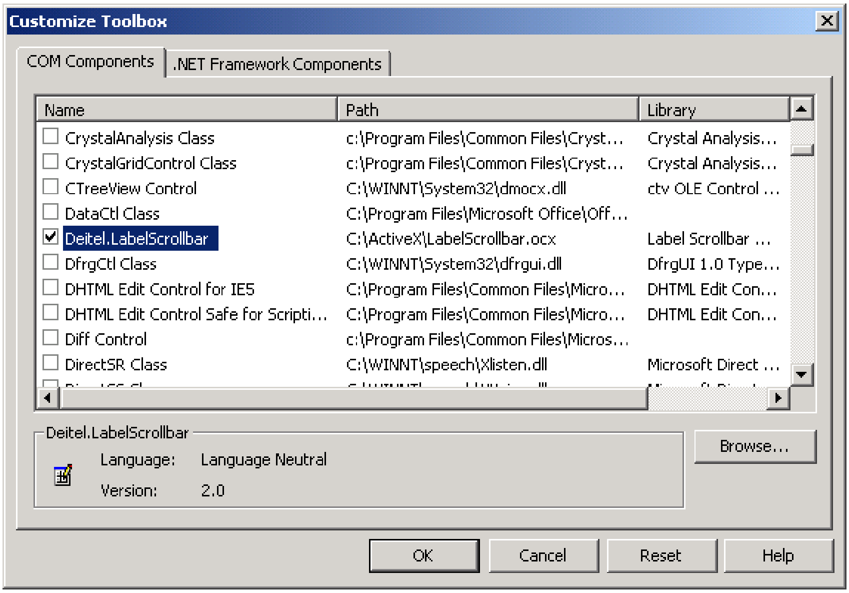
DllRegisterServer in C:\ActiveX\LabelScrollbar.ocx succeeded.

Fig. H.1 ActiveX control registration.

Once students register the ActiveX control, they must add it to the Visual Studio IDE’s toolbox. To accomplish this, right click the toolbox and select **Customize Toolbox** (Fig. H.2). The **Customize Toolbox** lists all the COM components that are registered on the machine. Select the **Deitel LabelScrollbar .OCX** file to add the **`LabelScrollbar`** to the toolbox and then click **OK**. When you add the **`LabelScrollBar`** to your Web form Visual Studio generates libraries **`Deitel`** and **`AxDeitel`** and adds them to the references in the **Solution Explorer**. The first reference (**`Deitel`**) is the *Common Runtime Library proxy*, which allows the programmer to access the ActiveX component’s methods and properties from .NET code. The second reference (**`AxDeitel`**) is the *Windows Forms proxy*, which allows the programmer to add the ActiveX component to a form. In this context a *proxy* is an object which allows .NET code to interact with COM code, we discuss the interaction between .NET and COM in greater detail in the next section. Note that, once the **`LabelScrollbar`** is added to the toolbox, two DLL files are generated in the **`bin`** directory of the application: **`AxInterop.Deitel.dll`** and **`Interop.Deitel.dll`**.

The first image in Fig. H.3 depicts the IDE toolbox after the **`LabelScrollBar`** has been added. The second image displays a list of the **`LabelScrollBar`** control properties, consisting of properties defined in the ActiveX control (i.e., **`Min`**, **`Max`**, **`SmallChange`**, **`LargeChange`**, and **`Value`**) and Visual Studio-defined properties (e.g., **`Anchor`**, **`Dock`** and **`Location`**).

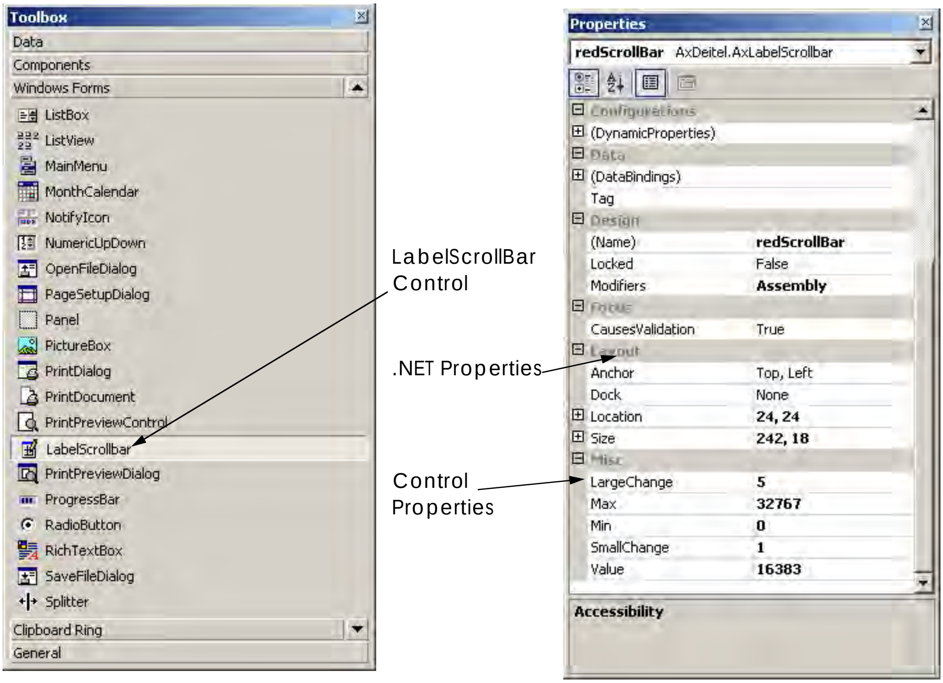
To demonstrate **`LabelScrollBar`**’s functionality, we add three **`LabelScrollbars`** to a form (Fig. H.4). These controls enable a user to select RGB values (i.e., red, green and blue) that specify the color of a **`PictureBox`**.



Name	Path	Library
☐ CrystalAnalysis Class	c:\Program Files\Common Files\Cryst...	Crystal Analysis...
☐ CrystalGridControl Class	c:\Program Files\Common Files\Cryst...	Crystal Analysis...
☐ CTreeView Control	C:\WINNT\System32\dmocx.dll	ctv OLE Control ...
☐ DataCtl Class	C:\Program Files\Microsoft Office\Off...	
☑ Deitel.LabelScrollbar	C:\ActiveX\LabelScrollbar.ocx	Label Scrollbar ...
☐ DfrgCtl Class	C:\WINNT\System32\dfrgui.dll	DfrgUI 1.0 Type...
☐ DHTML Edit Control for IE5	C:\Program Files\Common Files\Micro...	DHTML Edit Con...
☐ DHTML Edit Control Safe for Scripti...	C:\Program Files\Common Files\Micro...	DHTML Edit Con...
☐ Diff Control	c:\Program Files\Common Files\Micros...	
☐ DirectSR Class	C:\WINNT\speech\Xlisten.dll	Microsoft Direct ...

Deitel.LabelScrollbar
Language: Language Neutral
Version: 2.0

Fig. H.2 **Customize Toolbox** dialog with an ActiveX control selected.


Toolbox
Data
Components
Windows Forms
ListBox
ListView
MainMenu
MonthCalendar
NotifyIcon
NumericUpDown
OpenFileDialog
PageSetupDialog
Panel
PictureBox
PrintDialog
PrintDocument
PrintPreviewControl
LabelScrollbar
PrintPreviewDialog
ProgressBar
RadioButton
RichTextBox
SaveFileDialog
Splitter
Clipboard Ring
General
LabelScrollBar Control
.NET Properties
Control Properties
Properties
Configurations
Data
Design
Focus
Layout
Misc
Accessibility

redScrollBar AxDeitel.AxLabelScrollbar

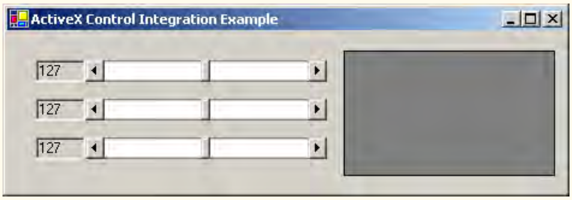
(DynamicProperties)	
(DataBindings)	
Tag	
(Name)	redScrollBar
Locked	False
Modifiers	Assembly
CausesValidation	True
Anchor	Top, Left
Dock	None
Location	24, 24
Size	242, 18
LargeChange	5
Max	32767
Min	0
SmallChange	1
Value	16383

Fig. H.3 IDE's toolbox and `LabelScrollbar` properties.

```
// Fig. H.4: LabelScrollbar.cs
// demonstrates ActiveX component usage in .NET.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using AxDeitel;

namespace FigH_04
{
   // Demonstrates ActiveX LabelScrollBar control
   public class LabelScrollTest : System.Windows.Forms.Form
   {
      private System.Windows.Forms.PictureBox pbColorBox;
      private AxDeitel.AxLabelScrollbar redScrollBar;
      private AxDeitel.AxLabelScrollbar greenScrollBar;
      private AxDeitel.AxLabelScrollbar blueScrollBar;

      /// Required designer variable.
      private System.ComponentModel.Container components = null;

```

Fig. H.4 ActiveX COM control integration in C#. (Part 1 of 3.)

```
      public LabelScrollTest()
      {
         InitializeComponent();

         // setup LabelScrollbar properties
         SetupRGBTitleScrollbar( redScrollBar );
         SetupRGBTitleScrollbar( blueScrollBar );
         SetupRGBTitleScrollbar( greenScrollBar );

         // initialize PictureBox back color
         pbColorBox.BackColor = Color.FromArgb(
            redScrollBar.Value, greenScrollBar.Value,
            blueScrollBar.Value );

      } // end constructor

      // initialize LabelScrollBar properties
      private void SetupRGBTitleScrollbar(
         AxLabelScrollbar scrollBar )
      {
         scrollBar.Min = 0;             // minimum value
         scrollBar.Max = 255;           // maximum value
         scrollBar.LargeChange = 10;    // large change value
         scrollBar.SmallChange = 1;     // small change value

      } // end method SetupRGBTitleScrollBar

      // Visual Studio .NET generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new LabelScrollTest() );
      }

      // handles scrollbar changed event
      private void scrollbar_Change( object sender,
         System.EventArgs e )
      {
         pbColorBox.BackColor = Color.FromArgb(
            redScrollBar.Value, greenScrollBar.Value,
            blueScrollBar.Value );
      } // end method scrollbar_Change
   } // end class LabelScrollTest
} // end namespace FigH_04
```



127

127

127

Fig. H.4 ActiveX COM control integration in C#. (Part 2 of 3.)

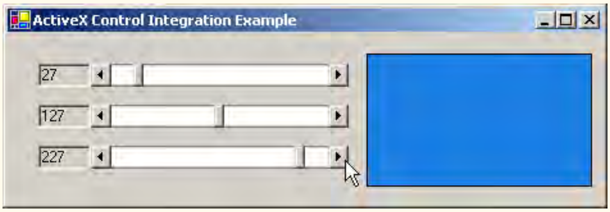


Fig. H.4 ActiveX COM control integration in C#. (Part 3 of 3.)

The constructor calls **`SetupRGBTitleScrollbar`** (lines 41–49), which sets the initial property values of **`Min`**, **`Max`**, **`LargeChange`** and **`SmallChange`** for each **`LabelScrollbar`** control. The **`Max`** property of each **`LabelScrollbar`** is set to **`255`**, allowing a color range of over 16 million colors.

The constructor also sets the **`PictureBox`**'s initial back color (lines 34–36). Lines 60–66 define the event handler for the **`LabelScrollbar`**'s **`change`** event. When the user changes the value of a **`LabelScrollbar`**, the **`change`** event fires and the **`PictureBox`**'s background color changes.

H.3 DLL Integration

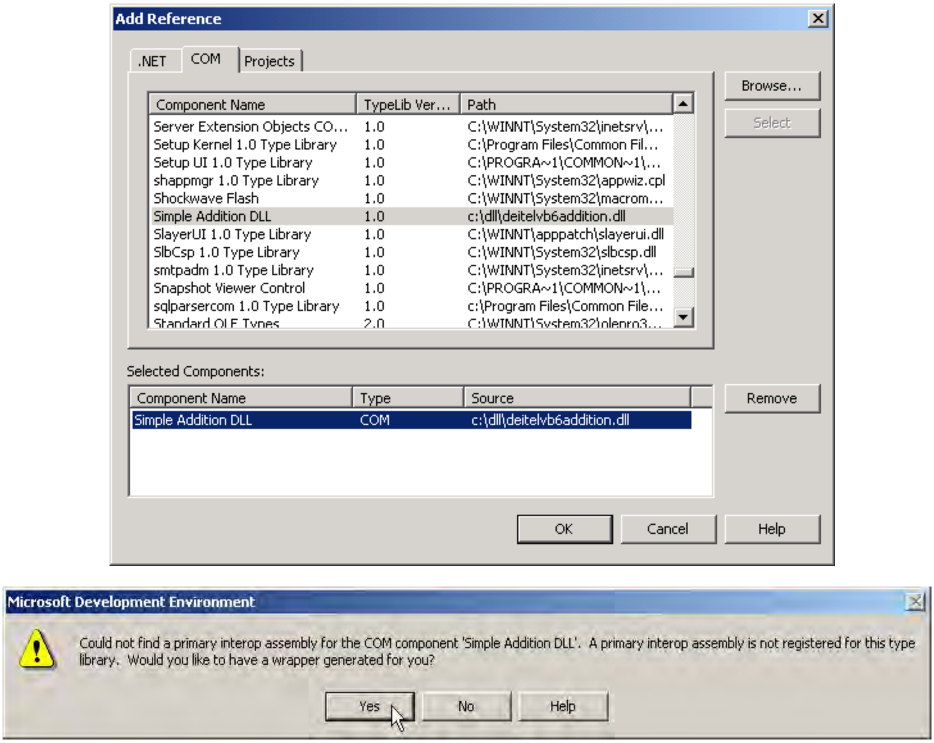
Visual Studio .NET also supports the integration of COM DLLs. This process is similar to the integration of ActiveX components. To demonstrate how Visual Studio .NET integrates a COM DLL, we have included **`deitelvb6addition.dll`** on the CD that accompanies this book. This simple library contains function **`AdditionFunction`**, which takes two arguments, adds them together and returns the result.

The first step in the integration of this COM DLL is to identify the DLL in the Windows Registry with the **`RegSvr32`** utility as follows:

```
regsvr32 deitelvb6addition.dll
```

After registering the library, add a reference to it in a C# program by right-clicking **References** in the **Solution Explorer** and selecting **Add Reference**. In the **Add Reference** dialog (Fig. H.5), select the **COM** tab. Then, choose **Simple Addition DLL** and click **OK**. A dialog appears, indicating that .NET must generate a *primary interop assembly* (Fig. H.5). Select **Yes** when this dialog appears. The primary interop assembly contains information for all methods and classes contained in the COM library.

In Windows, all components (both .NET and COM) must be accompanied by identification information. This information contains *Globally Unique Identifiers (GUID)* for the component and its internal classes, as well as language-independent descriptions of all classes and types that are defined in the component. These language-independent descriptions help to enable component reuse across multiple programming languages. A GUID consists of a long string of numbers that a computer generates on the basis of the computer's current time, hardware and a random number. The GUID algorithm never generates the same identifier twice; thus, GUIDs enable unique component identification across all computers.



Component Name	TypeLib Ver...	Path
Server Extension Objects CO...	1.0	C:\WINNT\System32\inetsrv\...
Setup Kernel 1.0 Type Library	1.0	C:\Program Files\Common Fil...
Setup UI 1.0 Type Library	1.0	C:\PROGRA~1\COMMON~1\...
shappmgr 1.0 Type Library	1.0	C:\WINNT\System32\appwiz.cpl
Shockwave Flash	1.0	C:\WINNT\System32\macrom...
Simple Addition DLL	1.0	c:\dll\deitelvb6addition.dll
SlayerUI 1.0 Type Library	1.0	C:\WINNT\apppatch\slayerui.dll
SlbCsp 1.0 Type Library	1.0	C:\WINNT\System32\slbcsp.dll
smtpadm 1.0 Type Library	1.0	C:\WINNT\System32\inetsrv\...
Snapshot Viewer Control	1.0	C:\PROGRA~1\COMMON~1\...
sqlparsercom 1.0 Type Library	1.0	c:\Program Files\Common File...
Standard OLE Types	2.0	C:\WINNT\System32\olepro3...

Component Name	Type	Source
Simple Addition DLL	COM	c:\dll\deitelvb6addition.dll

Could not find a primary interop assembly for the COM component 'Simple Addition DLL'. A primary interop assembly is not registered for this type library. Would you like to have a wrapper generated for you?

Fig. H.5 **Add Reference** dialog DLL Selection.

When a COM component is registered, its GUID is stored in the Windows Registry; programs then can use the registry to locate and identify the component. Once a program has located a desired component, it uses the component's *type library* to find and use the library's objects and methods. A *type library* describes all of a COM component's interfaces, types and methods; the type library is included in either the component **`.dll`** file or in a separate **`.tlb`** file. The separation of component identifiers (located in the Windows Registry) from the data represented on the disk (the library file) is the source of many problems associated with the COM architecture. By contrast, .NET components avoid these problems by maintaining all identification information internally. When Visual Studio imports a COM component, it creates a file that contains all identification and data-description information internally. Visual Studio obtains the component GUID from the Windows Registry and converts the data description from the type library-format into the .NET assembly format. This processing creates a new DLL file, called a primary interop assembly, which then is placed into the applications **`bin`** directory.

The primary interop assembly is used by .NET to locate COM methods and to translate component data types between the .NET platform types and COM component types. The translation for each COM component is performed by a *Runtime Callable Wrapper (RCW)*. The RCW is a proxy object created by the .NET runtime from the information in the object's primary interop assembly. The RCW manages the COM object and performs communication between .NET code and the COM object.

Performance Tip H.1

The .NET and COM architectures are fundamentally different in terms of memory management and object representation. Method calls to COM objects can degrade program performance because the RCW must convert all data types between the managed (.NET) context and the unmanaged (COM) context.

When we instantiate a COM object in C#, we are actually creating a new instance of the object's RCW. The communication between the RCW and the COM component is entirely transparent, enabling the .NET developer to interact with the COM object as if it were a .NET object.

We created an application (Fig. H.6) demonstrating how to use the **Simple Addition DLL** that we want to integrate into .NET. This program includes three text boxes and a button. After entering an `int` into each of the first two text boxes, the user clicks the **Calculate** button, and the program displays the sum of the two `int` in the third text box.

Line 29 creates the RCW `additionObject` for COM component `Deitel_DLL.CAddition`. Lines 55–59 enable the `calculateButton` button if both text boxes contain values, and disable the button if the text boxes are empty. When the user clicks **Calculate**, the button fires the event handler `calculateButton_Click`, which obtains the content of the text boxes and adds the values. The event handler calls COM method `addFunction`, which returns the sum as an `int` (lines 74–75). The result then is displayed in `txtResultBox` (line 77).

In this appendix, we demonstrated the use of COM libraries and ActiveX controls from a .NET application. In addition, we briefly explored the history of COM and the differences between its architecture and that of .NET. After reading this appendix, students should have a basic understanding of COM and should be able to use COM components in .NET applications. To learn more about .NET and COM, consult the Web resources described in Section H.4.

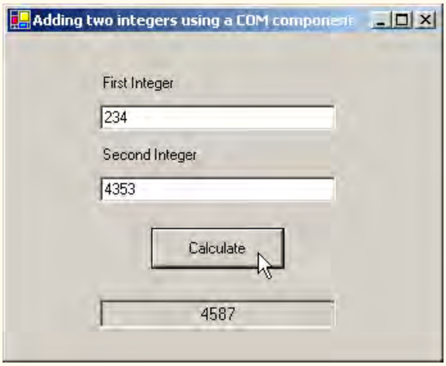
```
 1  // Fig. H.6: Addition.cs
 2  // Uses a COM component to add two integers.
 3
 4  using System;
 5  using System.Drawing;
 6  using System.Collections;
 7  using System.ComponentModel;
 8  using System.Windows.Forms;
 9  using System.Data;
10  using Deitel_DLL;
11
12  namespace Addition
13  {
14     // Adds two integers using a COM component
15     public class Addition : System.Windows.Forms.Form
16     {
17        // display labels
18        private System.Windows.Forms.Label SecondLabel;
19        private System.Windows.Forms.Label FirstLabel;
20
```

Fig. H.6 COM DLL component in C#. (Part 1 of 3.)

```
      // integer text boxes
      private System.Windows.Forms.TextBox resultBox;
      private System.Windows.Forms.TextBox firstIntegerBox;
      private System.Windows.Forms.TextBox secondIntegerBox;

      // calculates addition
      private System.Windows.Forms.Button calculateButton;

      private CAddition additionObject = new CAddition();

      // Required designer variable.
      private System.ComponentModel.Container
         components = null;

      public Addition()
      {
         InitializeComponent();
      }

      // Visual Studio .NET generated code

      // The main entry point for the application.
      [STAThread]
      static void Main()
      {
         Application.Run( new Addition() );
      }

      // event handler to enable calculateButton
      private void integerBox_TextChanged( object sender,
         System.EventArgs e )
      {
         // enable calculate button if both boxes
         // contain text
         if ( firstIntegerBox.Text != "" &&
            secondIntegerBox.Text != "" )
            calculateButton.Enabled = true;
         else
            calculateButton.Enabled = false;
      } // end method integerBox_TextChanged

      // event handler that displays sum when calculate
      // is clicked
      private void calculateButton_Click( object sender,
         System.EventArgs e )
      {
         int firstInteger, secondInteger, total;

         firstInteger = Int32.Parse( firstIntegerBox.Text );
         secondInteger = Int32.Parse( secondIntegerBox.Text );

         // addition object invokes AddFunction that
         // returns integer value
```

Fig. H.6 COM DLL component in C#. (Part 2 of 3.)

```
         total = additionObject.AddFunction( ref firstInteger,
            ref secondInteger );

         resultBox.Text = total.ToString();
      } // end method calculateButton_Click
   } // end class Addition
} // end namespace Addition
```


Adding two integers using a COM component

First Integer

234

Second Integer

4353

4587

Fig. H.6 COM DLL component in C#. (Part 3 of 3.)

H.4 Internet and World Wide Web Resources

`www.microsoft.com/com`

The Microsoft COM Web page provides technical white papers, documentation and developer support. This Web page is an essential resource for COM developers.

`www.cs.umd.edu/~pugh/com`

This Web site presents a high-level technical overview of the COM architecture.

`msdn.microsoft.com/msdnmag/issues/01/08/Interop/Interop.asp`

This Web site provides an introduction to integration services provided in .NET. The Web site includes introductory examples and describes .NETs COM Interopability capabilities.

SUMMARY

- Initially, applications created for Windows or DOS were designed as single monolithic executables—entire applications packaged in single executable files.
- As applications grew larger and more complex, it became impractical for developers to construct and distribute all the necessary components of an application, which resulted in longer development times and more costly distribution mechanism.
- Microsoft incorporated dynamic link libraries (DLLs) in Windows to allow developers to modularize and reuse code.
- A shared library, or dynamic link library, is a file containing compiled code that an application loads at execution time.
- Runtime loading allows developers to modify a single library and immediately test the results without rebuilding the entire application.
- Shared libraries increase the modularity of programs by allowing multiple applications to access a single code library.

- The partition of programs into smaller "pieces" makes it easier to distribute application upgrades, because only modified DLLs must be redistributed.
- Often, developers packaged DLLs with their applications to ensure that users were running the library version designed for their software. However, the packaged DLLs could overwrite preexisting libraries on users' systems, possibly breaking previously installed software.
- The problems introduced by shared libraries were so difficult to locate and fix that their effects became known as "DLL hell."
- In an attempt to combat "DLL hell," Microsoft developed the Component Object Model (COM).
- COM is a specification that controls library versions, backwards compatibility and language interoperability.
- The COM specification, defined by Microsoft, is detailed and strict to ensure that COM developers create compatible libraries.
- Microsoft implemented the COM architecture on a large scale. Today, virtually all Windows libraries adhere to the COM specification.
- When implemented correctly, COM ensures highly organized and reusable libraries, but it does have limitations.
- COM is difficult to program and deploy, because developers must guarantee that new COM components are both compatible with previous versions and registered with the system.
- Microsoft .NET components retain the benefits of COM while avoiding many of its associated problems.
- To facilitate developers migration from COM to .NET, Microsoft created a set of utilities designed to integrate existing COM components into the .NET platform.
- ActiveX controls are a commonly used COM component.
- The `RegSvr32` utility program registers COM components with the operating system.
- ActiveX can be imported into the IDE's toolbox.
- The **Customize Toolbox** option lists all the COM components that are registered on a specific machine.
- Windows components contain Globally Unique Identifiers (GUID) for the component and its internal classes as well as language-independent descriptions of all classes and types that are defined in the component.
- Language-independent descriptions help to enable component reuse across multiple programming languages.
- A GUID consists of a long string of numbers that a computer generates on the basis of the computer's current time, hardware and a random number. The GUID algorithm never generates the same identifier twice; thus, GUIDs enable unique component identification across all computers.
- When a COM component is registered, its GUID is stored in the Windows Registry; programs then can use the registry to locate and identify the component.
- A *type library* describes all of a COM component's interfaces, types and methods; the type library is included in either the component `.dll` file or in a separate `.tlb` file.
- The separation of component identifiers (located in the Windows Registry) from the data represented on the disk (the library file) is the source of many problems associated with the COM architecture.
- .NET components problems by maintaining all identification information internally.
- A primary interop assembly is used by .NET to locate COM methods and to translate component data types between the .NET platform types and COM component types.

- The translation for each COM component is performed by a Runtime Callable Wrapper (RCW). The RCW is a proxy object created by the .NET runtime from the information in the object's primary interop assembly.
- The RCW manages the COM object and performs communication between .NET code and the COM object.
- When we instantiate a COM object in C#, we are actually creating a new instance of the object's RCW. The communication between the RCW and the COM component is entirely transparent, enabling the .NET developer to interact with the COM object as if it were a .NET object.

TERMINOLOGY

ActiveX
COM (Component Object Model)
COM component
COM limitation
Common Runtime Library proxy
Component Object Model (COM)
DLL (Dynamic Link Library)
`.dll` file
DLL hell
Dynamic Link Library (DLL)
Globally Unique Identifier (GUID)
GUID (Globally Unique Identifier)
monolithic executable
.NET component
`.OCX` file
primary interop assembly
proxy
RCW (Runtime Callable Wrapper)
`RegSvr32` utility
Runtime Callable Wrapper (RCW)
shared library
`.tlb` file
Windows Forms proxy
Windows Registry

SELF-REVIEW EXERCISES

H.1 Fill in the blanks in each of the following statements:

a) Initially, DOS and Windows programs were designed as __________ executables.

b) Microsoft incorporated shared libraries, or __________, into Windows to allow program modularity and code reusability.

c) The COM specification was designed as a uniform programming model that promotes __________, __________ and __________.

d) A __________ contains language-independent descriptions of all interfaces, methods and data types defined in a COM component.

e) When an ActiveX control is imported, two files are created: A(n) __________ proxy and __________ proxy.

H.2 State whether each of the following statements is *true* or *false*. If *false*, explain why.

a) The Runtime Callable Wrapper (RCW) allows .NET components to be accessed from COM components.

b) A primary interop assembly contains information about COM components.

c) .NET component GUIDs must be entered in the Windows Registry.

d) The Component Object Model eliminates "DLL hell."

e) ActiveX controls are COM components.

ANSWERS TO SELF-REVIEW EXERCISES

H.1 a) monolithic. b) Dynamic Link Libraries (DLLs). c) language independence, backwards compatibility and version control. d) type library e) Common Language Runtime, Windows Forms proxy.

H.2 a) False. The RCW allows .NET components to access COM components. b) True. c) False. Unlike COM components, .NET components contain their GUIDs internally. d) False. COM DLLs may not be correctly representing in the windows registry causing misidentification and resulting in "DLL hell." e) True.

7

Introduction to HyperText Markup Language 4: Part 1

Objectives

- To understand the key components of an HTML document.
- To be able to use basic HTML elements to create World Wide Web pages.
- To be able to add images to your Web pages.
- To understand how to create and use hyperlinks to traverse Web pages.
- To be able to create lists of information.

To read between the lines was easier than to follow the text.
Henry James

Mere colour, unspoiled by meaning, and annulled with definite form, can speak to the soul in a thousand different ways.
Oscar Wide

High thoughts must have high language.
Aristophanes

I've gradually risen from lower-class background to lower-class foreground.
Marvin Cohen

Outline

I.1 Introduction
I.2 Markup Languages
I.3 Editing HTML
I.4 Common Elements
I.5 Headers
I.6 Linking
I.7 Images
I.8 Special Characters and More Line Breaks
I.9 Unordered Lists
I.10 Nested and Ordered Lists
I.11 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

I.1 Introduction

In this appendix we introduce the basics of creating Web pages in HTML. We write many simple Web pages. In Appendix J, Introduction to HyperText Markup Language 4: Part 2, we introduce more sophisticated HTML techniques, such as *tables*, which are particularly useful for structuring information from databases. In this appendix, we do not present any C# programming.

In this appendix, we introduce basic HTML *elements* and *attributes*. A key issue when using HTML is the separation of the *presentation of a document* (i.e., how the document is rendered on the screen by a browser) from the *structure of that document*. In this appendix and in Appendix J, we discuss this issue in depth.

I.2 Markup Languages

HTML is a *markup language*. It is used to format text and information. This “marking up” of information is different from the intent of traditional programming languages, which is to perform actions in a designated order.

In HTML, text is marked up with *elements*, delineated by *tags* that are keywords contained in pairs of angle brackets. For example, the HTML *element* itself, which indicates that we are writing a Web page to be rendered by a browser, begins with the start tag `<html>` and terminates with the end tag `</html>`. These elements format your page in a specified way. Over the course of the next two appendices, we introduce many of the commonly used tags and how to use them.

Good Programming Practice I.1

HTML tags are not case sensitive. However, keeping all the letters in one case improves program readability. Although the choice of case is up to you, we recommend that you write all of your code in lowercase. Writing in lowercase ensures greater compatibility with future markup languages that are designed to be written with only lowercase tags and elements.

Common Programming Error I.1

Forgetting to include end tags for elements that require them is a syntax error and can grossly affect the formatting and look of your page. Unlike conventional programming languages, a syntax error in HTML does not usually cause page display in browsers to fail completely.

I.3 Editing HTML

In this appendix we show how to write HTML in its *source-code form*. We create *HTML documents* using a text editor and store them in files with either the `.html` or `.htm` file name extension. A wide variety of text editors exist. We recommend that you initially use a text editor called Notepad, which is built into Windows. Notepad can be found inside the **Accessories** panel of your **Program** list, inside the **Start** menu. You can also download a free HTML source-code editor called HTML-Kit at `www.chami.com/html-kit`. Unix users can use popular text editors like *vi* or *emacs*.

Good Programming Practice I.2

Assign names to your files that describe their functionality. This practice can help you identify documents faster. It also helps people who want to link to your page, by giving them an easier-to-remember name for the file. For example, if you are writing an HTML document that will display your products, you might want to call it `products.html`.

As mentioned previously, errors in conventional programming languages like C, C++ and Visual Basic often prevent the program from running. Errors in HTML markup are usually not fatal. The browser will make its best effort at rendering the page, but will probably not display the page as you intended.

The file name of your *home page* (the first of your HTML pages that a user sees when browsing your Web site) should be `index.html`, because when a browser does not request a specific file in a directory, the normal default Web server response is to return `index.html` (this may be different for your server) if it exists in that directory. For example, if you direct your browser to `www.deitel.com`, the server actually sends the file `www.deitel.com/index.html` to your browser.

I.4 Common Elements

Throughout these HTML appendices, we will present both HTML source code and a sample screen capture of the rendering of that HTML in Internet Explorer. Figure I.1 shows an HTML file that displays one line of text.

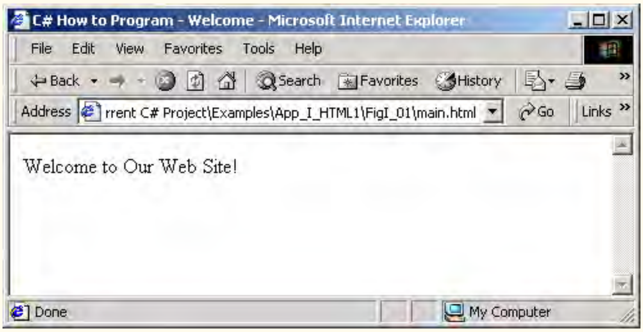
Lines 1 and 2

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
           "http://www.w3.org/TR/html4/strict.dtd">
```

are required in every HTML document and are used to specify the *document type*. The document type specifies which version of HTML is used in the document and can be used with a validation tool, such as the W3C's `validator.w3.org`, to ensure an HTML document conforms to the HTML recommendation. In these examples we create HTML version 4.01 documents. All of the examples in these appendices have been validated through the Web site `validator.w3.org`.

The HTML document begins with the opening `<html>` tag (line 3) and ends with the closing `</html>` tag (line 17).

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2            "http://www.w3.org/TR/html4/strict.dtd">
3  <html>
4
5  <!-- Fig. I.1: main.html -->
6  <!-- Our first Web page. -->
7
8  <head>
9     <title>C# How to Program - Welcome</title>
10 </head>
11
12 <body>
13
14    <p>Welcome to Our Web Site!</p>
15
16 </body>
17 </html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
rrent C# Project\Examples\App_I_HTML1\FigI_01\main.html
Go
Links
Done
My Computer

Welcome to Our Web Site!

Fig. I.1 Basic HTML file.

Good Programming Practice I.3

Always include the `<html>`*…*`</html>` *tags in the beginning and end of your HTML document.*

Good Programming Practice I.4

Place comments throughout your code. Comments in HTML are placed inside the `<!--`*…*`-->` *tags. Comments help other programmers understand the code, assist in debugging and list other useful information that you do not want the browser to render. Comments also help you understand your own code, especially if you have not looked at it for a while.*

We see our first *comments* (i.e., text that documents or describes the HTML markup) on lines 5 and 6

```
<!-- Fig. I.1: main.html -->
<!-- Our first Web page. -->
```

Comments in HTML always begin with `<!--` and end with `-->`. The browser ignores any text and/or tags inside a comment. We place comments at the top of each HTML document giving the figure number, the file name and a brief description of the purpose of the example. In subsequent examples, we also include comments in the markup, especially when we introduce new features.

Every HTML document contains a ***`head`*** element, which generally contains information about the document, and a `body` element, which contains the page content. Information in the `head` element is not generally rendered in the display window, but may be made available to the user through other means. Lines 8–10

```
<head>
   <title>C# How to Program - Welcome</title>
</head>
```

show the `head` element section of our Web page. Including a ***`title`*** element is required for every HTML document. To include a title in your Web page, enclose your chosen title between the pair of tags ***`<title>`***…***`</title>`*** in the `head` element.

Good Programming Practice I.5

Use a consistent title-naming convention for all pages on your site. For example, if your site is called "Al's Web Site," then the title of your links page might best be "Al's Web Site - Links". This practice presents a clearer picture to those browsing your site.

The `title` element names your Web page. The title usually appears on the colored bar at the top of the browser window, and also will appear as the text identifying your page if a user adds your page to their list of **Favorites** or **Bookmarks**. The title is also used by search engines for cataloging purposes, so picking a meaningful title can help search engines direct a more focused group of people to your site.

Line 12

```
<body>
```

opens the ***`body`*** element. The body of an HTML document is the area where you place the content of your document. This includes text, images, links and forms. We discuss many elements that can be inserted in the `body` element later in this appendix. Remember to include the end `</body>` tag before the closing `</html>` tag.

Various elements enable you to place text in your HTML document. We see the *paragraph element* on line 14

```
<p>Welcome to Our Web Site!</p>
```

All text placed between the ***`<p>`***…***`</p>`*** tags forms one paragraph. Most Web browsers render paragraphs as set apart from all other material on the page by a line of vertical space both before and after the paragraph. The HTML in line 12 causes Internet Explorer to render the enclosed text as shown in Fig. I.1.

Our code example ends on lines 16 and 17 with

```
</body>
</html>
```

These two tags close the body and HTML sections of the document, respectively. As discussed earlier, the last tag in any HTML document should be `</html>`, which tells the browser that all HTML coding is complete. The closing `</body>` tag is placed before the `</html>` tag because the body section of the document is entirely enclosed by the HTML section. Therefore, the body section must be closed before the HTML section.

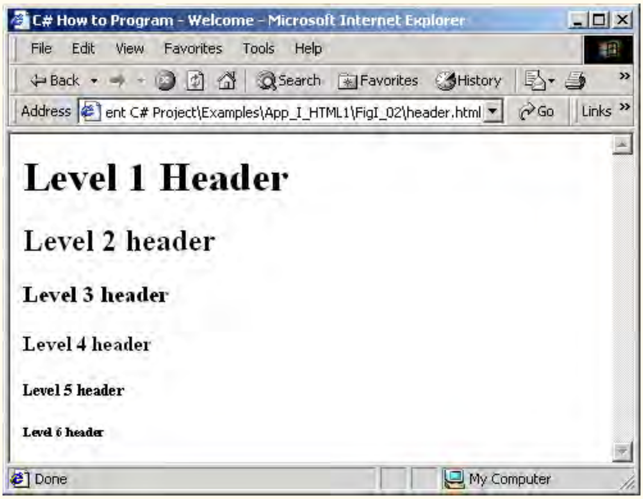
I.5 Headers

The six *headers* are used to delineate new sections and subsections of a page. Figure I.2 shows how these elements (**h1** through **h6**) are used. Note that the actual size of the text of each header element is selected by the browser and can vary significantly between browsers.

Good Programming Practice I.6

Adding comments to the right of short HTML lines is a clean-looking way to comment code.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.2: header.html -->
<!-- HTML headers.         -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

<body>

   <h1>Level 1 Header</h1>   <!-- Level 1 header -->
   <h2>Level 2 header</h2>   <!-- Level 2 header -->
   <h3>Level 3 header</h3>   <!-- Level 3 header -->
   <h4>Level 4 header</h4>   <!-- Level 4 header -->
   <h5>Level 5 header</h5>   <!-- Level 5 header -->
   <h6>Level 6 header</h6>   <!-- Level 6 header -->

</body>
</html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address ent C# Project\Examples\App_I_HTML1\FigI_02\header.html
Go Links
Done
My Computer

Level 1 Header

Level 2 header

Level 3 header

Level 4 header

Level 5 header

Level 6 header

Fig. I.2 Header elements **h1** through **h6**.

Line 14

```
<h1>Level 1 Header</h1>
```

introduces the ***h1*** *header element*, with its start tag ***<h1>*** and its end tag ***</h1>***. Any text to be displayed is placed between the two tags. All six header elements, **h1** through **h6**, follow the same pattern.

Good Programming Practice I.7

Putting a header at the top of every Web page helps those viewing your pages understand what the purpose of each page is.

I.6 Linking

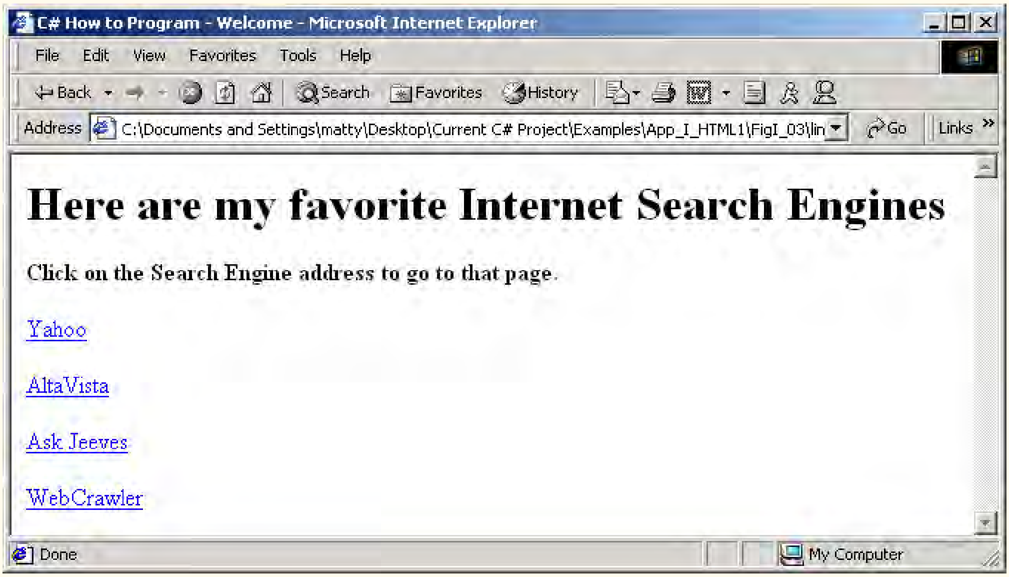
The most important capability of HTML is its ability to create hyperlinks to other documents, making possible a worldwide network of linked documents and information. In HTML, both text and images can act as *anchors* to *link* to other pages on the Web. We introduce anchors and links in Fig. I.3.

The first link can be found on line 19

```
<p><a href = "http://www.yahoo.com">Yahoo</a></p>
```

```
 1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 2            "http://www.w3.org/TR/html4/strict.dtd">
 3  <html>
 4
 5  <!-- Fig. I.3: links.html        -->
 6  <!-- Introduction to hyperlinks. -->
 7
 8  <head>
 9     <title>C# How to Program - Welcome</title>
10  </head>
11
12  <body>
13
14     <h1>Here are my favorite Internet Search Engines</h1>
15
16     <p><strong>Click on the Search Engine address to go to that
17        page.</strong></p>
18
19     <p><a href = "http://www.yahoo.com">Yahoo</a></p>
20
21     <p><a href = "http://www.altavista.com">AltaVista</a></p>
22
23     <p><a href = "http://www.askjeeves.com">Ask Jeeves</a></p>
24
25     <p><a href = "http://www.webcrawler.com">WebCrawler</a></p>
26
27  </body>
28  </html>
```

Fig. I.3 Linking to other Web pages. (Part 1 of 2.)


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_I_HTML1\FigI_03\lin
Go Links
Done
My Computer

Here are my favorite Internet Search Engines

Click on the Search Engine address to go to that page.

Yahoo

AltaVista

Ask Jeeves

WebCrawler

Fig. I.3 Linking to other Web pages. (Part 2 of 2.)

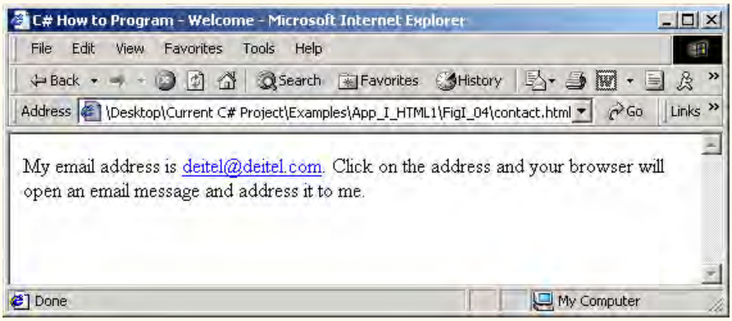
Links are inserted with the **`a`** *(anchor) element*. The anchor element is unlike the elements we have seen thus far in that it requires certain *attributes* (i.e., markup that provides information about the element) to specify the hyperlink. Attributes are placed inside an element's start tag and consist of a name and a value. The most important attribute for the **`a`** element is the location to which you would like the anchoring object to be linked. This location can be any resource on the Web, including pages, files and email addresses. To specify the address to link to, add the **`href`** *attribute* to the anchor element as follows: **`<a href = "`***address***`">`**. In this case, the address we are linking to is **`http://www.yahoo.com`**. The hyperlink (line 19) makes the text **`Yahoo`** a link to the address specified in **`href`**.

Anchors can use **`mailto`** URLs to provide links to email addresses. When someone selects this type of anchored link, most browsers launch the default email program to initiate an email message to the linked address. This type of anchor is demonstrated in Fig. I.4.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.4: contact.html   -->
<!-- Adding email hyperlinks. -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

```

Fig. I.4 Linking to an email address. (Part 1 of 2.)

```
12  <body>
13
14     <p>My email address is <a href = "mailto:deitel@deitel.com">
15     deitel@deitel.com</a>. Click on the address and your browser
16     will open an email message and address it to me.</p>
17
18  </body>
19  </html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
Done
My Computer

\Desktop\Current C# Project\Examples\App_I_HTML1\FigI_04\contact.html

My email address is deitel@deitel.com. Click on the address and your browser will open an email message and address it to me.

Fig. I.4 Linking to an email address. (Part 2 of 2.)

We see an email link on lines 14 and 15

```
<p>My email address is <a href = "mailto:deitel@deitel.com">
deitel@deitel.com</a>. Click on the address and your browser
```

The form of an email anchor is `<a href = "mailto:`*emailaddress*`">`…`</a>`. It is important that this whole attribute, including the `mailto:`, be placed in quotation marks.

I.7 Images

We have thus far dealt exclusively with text. We now show how to incorporate images into Web pages (Fig. I.5).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.5: picture.html   -->
<!-- Adding images with HTML. -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

```

Fig. I.5 Placing images in HTML files. (Part 1 of 2.)

```
12  <body>
13
14     <p><img src = "csphtp.jpg" height = "236" width = "181"
15        alt = "Demonstration of the alt attribute"></p>
16
17  </body>
18  </html>
```

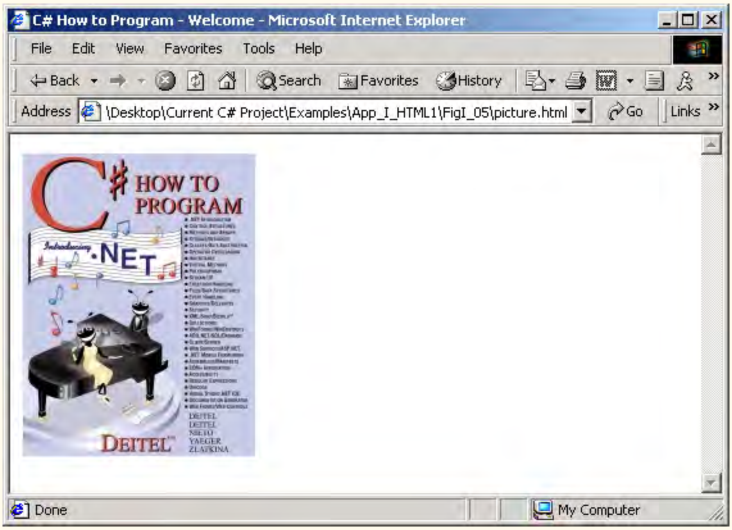

C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address \Desktop\Current C# Project\Examples\App_I_HTML1\FigI_05\picture.html
Go Links
C# HOW TO PROGRAM
Introducing .NET
DEITEL
Done
My Computer

Fig. I.5 Placing images in HTML files. (Part 2 of 2.)

The image in this code example is inserted in lines 14 and 15:

```
<p><img src = "csphtp.jpg" height = "236" width = "181"
   alt = "Demonstration of the alt attribute"></p>
```

You specify the location of the image file in the ***img*** element. This is done by adding the **`src = "`**_location_**`"`** attribute. You can also specify the ***height*** and ***width*** of an image, measured in pixels. The term pixel stands for “picture element.” Each pixel represents one dot of color on the screen. This image is 181 pixels wide and 236 pixels high.

Good Programming Practice I.8

*Always include the **height** and the **width** of an image inside the **img** tag. When the browser loads the HTML file, it will know immediately how much screen space to give the image and will therefore lay out the page properly, even before it downloads the image.*

Common Programming Error I.2

Entering new dimensions for an image that changes its inherent width-to-height ratio distorts the appearance of the image. For example, if your image is 200 pixels wide and 100 pixels high, you should always make sure that any new dimensions have a 2:1 width-to-height ratio.

The ***alt*** attribute is required for every **`img`** element. In Fig. I.5, the value of this attribute is

```
alt = "Demonstration of the alt attribute"
```

Attribute **`alt`** is provided for browsers that have images turned off or cannot view images (e.g., text-based browsers). The value of the **`alt`** attribute will appear on-screen in place of the image, giving the user an idea of what was in the image. The **`alt`** attribute is especially important for making Web pages *accessible* to users with disabilities, as discussed in Chapter 24, Accessibility.

Good Programming Practice I.9

Include a description of the purpose of every image, using the **`alt`** *attribute in the* **`img`** *tag.*

Now that we have discussed placing images on your Web page, we will show you how to transform images into anchors to provide links to other sites on the Internet (Fig. I.6).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.6: nav.html            -->
<!-- Using images as link anchors. -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

<body>

   <p>
      <a href = "links.html">
      <img src = "buttons/links.jpg" width = "65" height = "50"
         alt = "Links Page"></a><br>

      <a href = "list.html">
      <img src = "buttons/list.jpg" width = "65" height = "50"
         alt = "List Example Page"></a><br>

      <a href = "contact.html">
      <img src = "buttons/contact.jpg" width = "65" height = "50"
         alt = "Contact Page"></a><br>

      <a href = "header.html">
      <img src = "buttons/header.jpg" width = "65" height = "50"
         alt = "Header Page"></a><br>

      <a href = "table.html">
      <img src = "buttons/table.jpg" width = "65" height = "50"
         alt = "Table Page"></a><br>

      <a href = "form.html">
      <img src = "buttons/form.jpg" width = "65" height = "50"
         alt = "Feedback Form"></a><br>
   </p>

```

Fig. I.6 Using images as link anchors. (Part 1 of 2.)

```
40  </body>
41  </html>
```

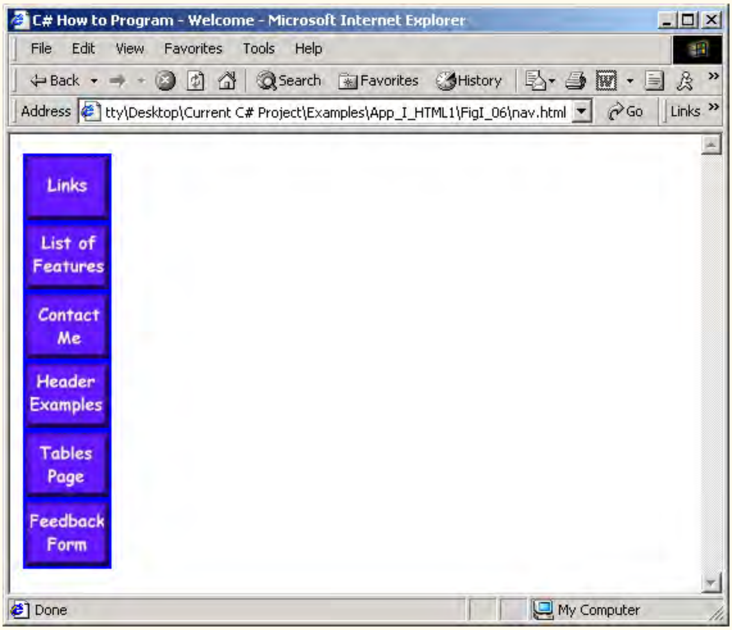

C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
tty\Desktop\Current C# Project\Examples\App_I_HTML1\FigI_06\nav.html
Go
Links
Links
List of Features
Contact Me
Header Examples
Tables Page
Feedback Form
Done
My Computer

Fig. I.6 Using images as link anchors. (Part 2 of 2.)

We see an image hyperlink in lines 15–17

```
<a href = "links.html">
<img src = "buttons/links.jpg" width = "65" height = "50"
   alt = "Links Page"></a><br>
```

Here we use the `a` element and the `img` element. The anchor works the same way as when it surrounds text; the image becomes an active hyperlink to a location somewhere on the Internet, indicated by the `href` attribute inside the `<a>` tag. Remember to close the anchor element when you want the hyperlink to end.

If you direct your attention to the `src` attribute of the `img` element,

```
src = "buttons/links.jpg"
```

you will see that it is not in the same form as that of the image in the previous example. This is because the image we are using here, `about.jpg`, resides in a subdirectory called `buttons`, which is in the main directory for our site. We have done this so that we can keep all our button graphics in the same place, making them easier to find and edit.

You can always refer to files in different directories simply by putting the directory name in the correct format in the `src` attribute. If, for example, there was a directory inside the `buttons` directory called `images`, and we wanted to put a graphic from that directory onto our page, we would just have to make the source attribute reflect the location of the image: `src = "buttons/images/filename"`.

You can even insert an image from a different Web site into your site (after obtaining permission from the site's owner, of course). Just make the `src` attribute reflect the location and name of the image file.

On line 17

```
alt = "Links Page"></a><br>
```

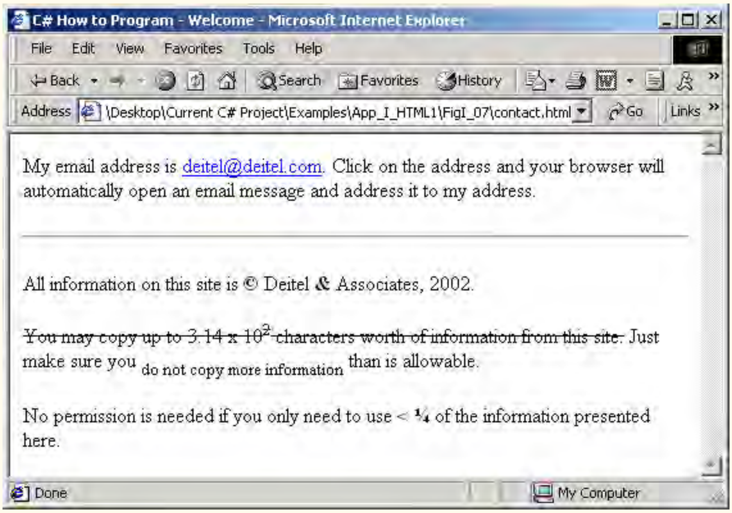
we introduce the ***br** element*, which causes a *line break* to be rendered in most browsers.

I.8 Special Characters and More Line Breaks

In HTML, the old QWERTY typewriter setup no longer suffices for all our textual needs. HTML 4.01 has a provision for inserting special characters and symbols (Fig. I.7).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.7: contact.html        -->
<!-- Inserting special characters. -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

<body>

   <!-- special characters are entered using the form &code; -->
   <p>My email address is <a href = "mailto:deitel@deitel.com">
   deitel@deitel.com</a>. Click on the address and your browser
   will automatically open an email message and address it to my
   address.</p>

   <hr> <!-- inserts a horizontal rule -->

   <p>All information on this site is <strong>&copy;</strong>
   Deitel <strong>&amp;</strong> Associates, 2002.</p>

   <!-- text can be struck out with a set of <del>...</del>   -->
   <!-- tags, it can be set in subscript with <sub>...</sub>, -->
   <!-- and it can be set into superscript with <sup...</sup> -->
   <p><del>You may copy up to 3.14 x 10<sup>2</sup> characters
   worth of information from this site.</del> Just make sure
   you <sub>do not copy more information</sub> than is allowable.
   </p>

   <p>No permission is needed if you only need to use <strong>
   &lt; &frac14;</strong> of the information presented here.</p>

</body>
</html>
```

Fig. I.7 Inserting special characters into HTML. (Part 1 of 2.)


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address \Desktop\Current C# Project\Examples\App_I_HTML1\FigI_07\contact.html
Go
Links
Done
My Computer

My email address is deitel@deitel.com. Click on the address and your browser will automatically open an email message and address it to my address.

All information on this site is © Deitel & Associates, 2002.

~~You may copy up to 3.14 x 10^2 characters worth of information from this site.~~ Just make sure you $_{\text{do not copy more information}}$ than is allowable.

No permission is needed if you only need to use < ¼ of the information presented here.

Fig. I.7 Inserting special characters into HTML. (Part 2 of 2.)

There are some *special characters* inserted into the text of lines 22 and 23:

```
<p>All information on this site is <strong>&copy;</strong>
Deitel <strong>&amp;</strong> Associates, 2002.</p>
```

All special characters are inserted in their code form. The format of the code is always **&*code*;**. An example of this is **&amp;**, which inserts an ampersand. Codes are often abbreviated forms of the character (like **amp** for ampersand and **copy** for copyright) and can also be in the form of *hex codes*. (For example, the hex code for an ampersand is 38, so another method of inserting an ampersand is to use **&#38;**.) Please refer to the chart in Appendix M for a listing of special characters and their respective codes.

In lines 28–31, we introduce three new styles.

```
<p><del>You may copy up to 3.14 x 10<sup>2</sup> characters
worth of information from this site.</del> Just make sure
you <sub>do not copy more information</sub> than is allow-
able.
</p>
```

You can indicate text that has been deleted from a document by including it in a ***del*** element. This could be used as an easy way to communicate revisions of an online document. Many browsers render the **del** element as strike-through text. To turn text into *superscript* (i.e., raised vertically to the top of the line and made smaller) or to turn text into *subscript* (the opposite of superscript, lowers text on a line and makes it smaller), use the ***sup*** or ***sub*** element, respectively.

Line 20

```
<hr> <!-- inserts a horizontal rule -->
```

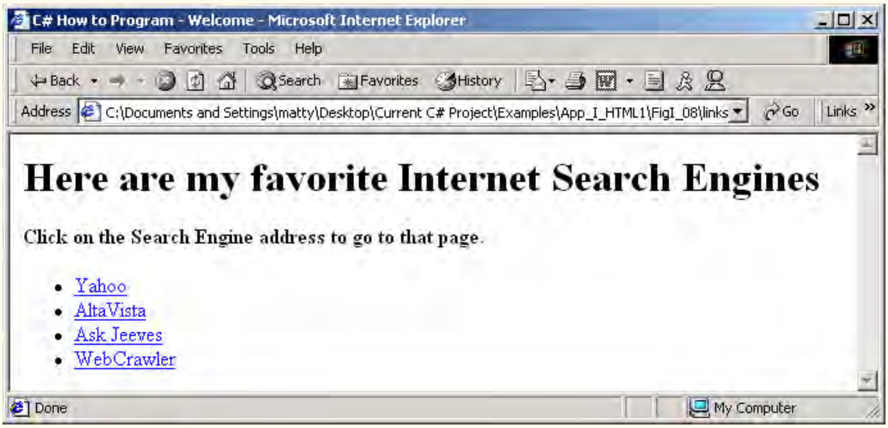
inserts a horizontal rule, indicated by the `<hr>` tag. A horizontal rule is rendered by most browsers as a straight line going across the screen horizontally. The `hr` element also inserts a line break directly below it.

I.9 Unordered Lists

Figure I.8 demonstrates displaying text in an *unordered list*. Here, we reuse the HTML file from Fig. I.3, adding an unordered list to enhance the structure of the page. The *unordered list element* `ul` creates a list in which every line begins with a bullet mark in most Web browsers.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.8: links.html                  -->
<!-- Unordered list containing hyperlinks. -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

<body>

   <h1>Here are my favorite Internet Search Engines</h1>


   <p><strong>Click on the Search Engine address to go to that
      page.</strong></p>

   <ul>
      <li>
         <a href = "http://www.yahoo.com">Yahoo</a>
      </li>

      <li>
         <a href = "http://www.altavista.com">AltaVista</a>
      </li>

      <li>
         <a href = "http://www.askjeeves.com">Ask Jeeves</a>
      </li>

      <li>
         <a href = "http://www.webcrawler.com">WebCrawler</a>
      </li>
   </ul>

</body>
</html>
```

Fig. I.8 Unordered lists in HTML. (Part 1 of 2.)


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_I_HTML1\FigI_08\links
Go Links
Done
My Computer

Here are my favorite Internet Search Engines

Click on the Search Engine address to go to that page.

- Yahoo
- AltaVista
- Ask Jeeves
- WebCrawler

Fig. I.8 Unordered lists in HTML. (Part 2 of 2.)

The first list item appears in lines 21–23

```
<li>
   <a href = "http://www.yahoo.com">Yahoo</a>
</li>
```

Each entry in an unordered list is a `li` (*list item*) element. Most Web browsers render these elements with a line break and a bullet mark at the beginning of the line.

I.10 Nested and Ordered Lists

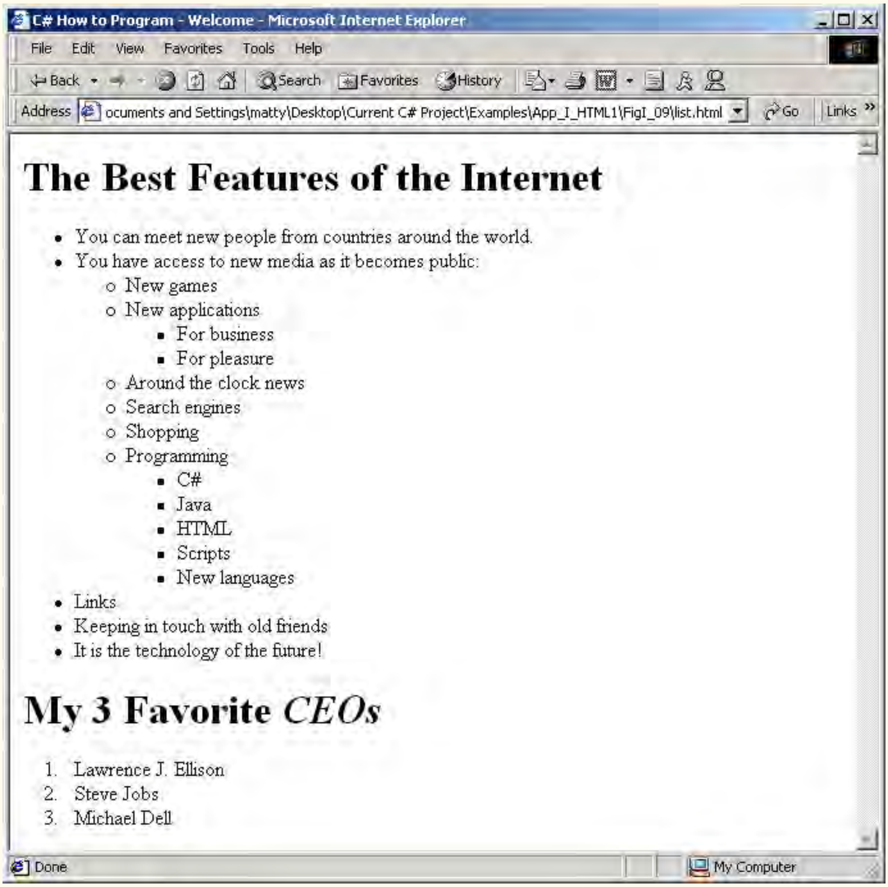
Figure I.9 demonstrates *nested lists* (i.e., one list inside another list). This feature is useful for displaying information in outline form.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. I.9: list.html                 -->
<!-- Advanced Lists: nested and ordered. -->

<head>
   <title>C# How to Program - Welcome</title>
</head>

<body>

   <h1>The Best Features of the Internet</h1>

   <ul>
      <li>You can meet new people from countries around
         the world.</li>
```

Fig. I.9 Nested and ordered lists in HTML. (Part 1 of 3.)

```
      <li>You have access to new media as it becomes public:

         <!-- this starts a nested list, which -->
         <!-- uses a modified bullet. The list -->
         <!-- ends when you close the <ul> tag -->
         <ul>
            <li>New games</li>
            <li>New applications

               <!-- another nested list -->
               <ul>
                  <li>For business</li>
                  <li>For pleasure</li>
               </ul> <!-- this ends the double nested list -->
            </li>

            <li>Around the clock news</li>
            <li>Search engines</li>
            <li>Shopping</li>
            <li>Programming

               <ul>
                  <li>C#</li>
                  <li>Java</li>
                  <li>HTML</li>
                  <li>Scripts</li>
                  <li>New languages</li>
               </ul>

            </li>

         </ul> <!-- this ends the first level nested list -->
      </li>

      <li>Links</li>
      <li>Keeping in touch with old friends</li>
      <li>It is the technology of the future!</li>

   </ul>   <!-- this ends the primary unordered list -->

   <h1>My 3 Favorite <em>CEOs</em></h1>

   <!-- ordered lists are constructed in the same way as   -->
   <!-- unordered lists, except their starting tag is <ol> -->
   <ol>
      <li>Lawrence J. Ellison</li>
      <li>Steve Jobs</li>
      <li>Michael Dell</li>
   </ol>

</body>
</html>
```

Fig. I.9 Nested and ordered lists in HTML. (Part 2 of 3.)


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address ocuments and Settings\matty\Desktop\Current C# Project\Examples\App_I_HTML1\FigI_09\list.html
Go Links
Done
My Computer

The Best Features of the Internet

- You can meet new people from countries around the world.
- You have access to new media as it becomes public:
  - New games
  - New applications
    - For business
    - For pleasure
  - Around the clock news
  - Search engines
  - Shopping
  - Programming
    - C#
    - Java
    - HTML
    - Scripts
    - New languages
- Links
- Keeping in touch with old friends
- It is the technology of the future!

My 3 Favorite *CEOs*

1. Lawrence J. Ellison
2. Steve Jobs
3. Michael Dell

Fig. I.9 Nested and ordered lists in HTML. (Part 3 of 3.)

Our first nested list begins on line 24, and its first element is on 25.

```
<ul>
   <li>New games</li>
```

A nested list is created in the same way as the list in Fig. I.8, except that the nested list is itself contained in a list element. Most Web browsers render nested lists by indenting the list one level and changing the bullet type for the list elements.

Good Programming Practice I.10

Indenting each level of a nested list in your code makes the code easier to edit and debug.

In Fig. I.9, lines 16–57 show a list with three levels of nesting. When nesting lists, be sure to insert the closing `</ul>` tags in the appropriate places. Lines 63–67

```
<ol>
   <li>Lawrence J. Ellison</li>
   <li>Steve Jobs</li>
   <li>Michael Dell</li>
</ol>
```

define an *ordered list* element with the tags `<ol>`…`</ol>`. Most browsers render ordered lists with a sequence number for each list element instead of a bullet. By default, ordered lists use decimal sequence numbers (1, 2, 3, …).

I.11 Internet and World Wide Web Resources

There are many resources available on the World Wide Web that go into more depth on the topics we cover. Visit the following sites for additional information on this appendix's topics.

`www.w3.org`
The *World Wide Web Consortium* (W3C), is the group that makes HTML recommendations. This Web site holds a variety of information about HTML—both its history and its present status.

`www.w3.org/TR/html401`
The *HTML 4.01 Specification* contains all the nuances and fine points in HTML 4.01.

`www.w3schools.com/html`
The HTMl School. This site contains a complete guide to HTML, starting with an introduction to the WWW and ending with advanced HTML features. This site also has a good reference for the features of HTML.

`www2.utep.edu/~kross/tutorial`
This University of Texas at El Paso site contains another guide for simple HTML programming. The site is helpful for beginners, because it focuses on teaching and gives specific examples.

`www.w3scripts.com/html`
This site, an offshoot of *W3Schools*, is a repository for code examples exhibiting all of the features of HTML, from beginner to advanced.

SUMMARY

- HTML is not a procedural programming language like C, Fortran, Cobol or Pascal. It is a markup language that identifies the elements of a page so a browser can render that page on the screen.
- HTML is used to format text and information. This "marking up" of information is different from the intent of traditional programming languages, which is to perform actions in a designated order.
- In HTML, text is marked up with elements, delineated by tags that are keywords contained in pairs of angle brackets.
- HTML documents are created via text editors.
- All HTML documents stored in files require either the **`.htm`** or the **`.html`** file name extension.
- Making errors while coding in conventional programming languages like C, C++ and Java often produces a fatal error, preventing the program from running. Errors in HTML code are usually not fatal. The browser will make its best effort at rendering the page, but will probably not display the page as you intended. In our Common Programming Errors and Testing and Debugging Tips, we highlight common HTML errors and how to detect and correct them.
- For most Web servers, the filename of your home page should be **`index.html`**. When a browser requests a directory, the default Web server response is to return **`index.html`**, if it exists in that directory.

- The document type specifies which version of HTML is used in the document and can be used with a validation tool, such as the W3C's **`validator.w3.org`**, to ensure an HTML document conforms to the HTML specification.
- **`<html>`** tells the browser that everything contained between the opening **`<html>`** tag and the closing **`</html>`** tag is HTML.
- Comments in HTML always begin with **`<!--`** and end with **`-->`** and can span across several source lines. The browser ignores any text and/or tags placed inside a comment.
- Every HTML file is separated into a header section and a body.
- Including a title is mandatory for every HTML document. Use the **`<title>`**…**`</title>`** tags to do so. They are placed inside the header.
- **`<body>`** opens the **`body`** element. The body of an HTML document is the area where you place all content you would like browsers to display.
- All text between the **`<p>`**…**`</p>`** tags forms one paragraph. Most browsers render paragraphs as set apart from all other material on the page by a line of vertical space both before and after the paragraph.
- Headers are a simple form of text formatting that typically increase text size based on the header's "level" (**`h1`** through **`h6`**). They are often used to delineate new sections and subsections of a page.
- The purpose of HTML is to mark up text; the question of how it is presented is left to the browser itself.
- People who have difficulty seeing can use special browsers that read the text on the screen aloud. These browsers (which are text based and do not show images, colors or graphics) might read **`strong`** and **`em`** with different inflections to convey the impact of the styled text to the user.
- You should close tags in the reverse order from that in which they were started to ensure proper nesting.
- The most important capability of HTML is creating hyperlinks to documents on any server to form a worldwide network of linked documents and information.
- Links are inserted with the **`a`** (anchor) element. To specify the address you would like to link to, add the **`href`** attribute to the anchor element, with the address as the value of **`href`**.
- Anchors can link to email addresses. When someone clicks this type of anchored link, their default email program initiates an email message to the linked address.
- The term pixel stands for "picture element". Each pixel represents one dot of color on the screen.
- You specify the location of the image file with the **`src = "`***location***`"`** attribute in the **`<img>`** tag. You can specify the **`height`** and **`width`** of an image, measured in pixels.
- **`alt`** is provided for browsers that cannot view pictures or that have images turned off (text-based browsers, for example). The value of the **`alt`** attribute will appear on screen in place of the image, giving the user an idea of what was in the image.
- You can refer to files in different directories by including the directory name in the correct format in the **`src`** attribute. You can insert an image from a different Web site onto your site (after obtaining permission from the site's owner). Just make the **`src`** attribute reflects the location and name of the image file.
- The **`br`** element forces a line break. If the **`br`** element is placed inside a text area, the text begins a new line at the place of the **`<br>`** tag.
- HTML 4.01 has a provision for inserting special characters and symbols. All special characters are inserted in the format of the code, always **`&`***code***`;`**. An example of this is **`&amp;`**, which inserts an ampersand. Codes are often abbreviated forms of the character (like **`amp`** for ampersand and **`copy`**

for copyright) and can also be in the form of hex codes. (For example, the hex code for an ampersand is 38, so another method of inserting an ampersand is to use **`&#38;`**.)

- The **`del`** element marks text as deleted, which is rendered with a strike through by most browsers. To turn text into superscript or subscript, use the **`sup`** or **`sub`** element, respectively.
- Most visual Web browsers place a bullet mark at the beginning of each element in an unordered list. All entries in an unordered list must be enclosed within **`<ul>`**…**`</ul>`** tags, which open and close the unordered list element.
- Each entry in an unordered list is contained in an **`li`** element. You then insert and format any text.
- Nested lists display information in outline form. A nested list is a list that is contained in an **`li`** element. Most visual Web browsers indent nested lists one level and change the bullet type to reflect the nesting.
- An ordered list (**`<ol>`**…**`</ol>`**) is rendered by most browsers with a sequence number instead of a bullet at the beginning of each list element. By default, ordered lists use decimal sequence numbers (1,2,3, …).

TERMINOLOGY

`&amp;`
`.htm`
`.html`
`<!--…-->` (comment)
`<body>`…**`</body>`**
`<hr>` element (horizontal rule)
`a` element (anchor; **`<a>`**…**`</a>`**)
`alt`
anchor
attributes of an HTML tag
`clear = "all"` in **`<br>`**
closing tag
color
comments
content of an HTML element
`del` element
`em` element (**`<em>`**…**`</em>`**)
emphasis
`h1` element (**`<h1>`**…**`</h1>`**)
`h2` element (**`<h2>`**…**`</h2>`**)
`h3` element (**`<h3>`**…**`</h3>`**)
`h4` element (**`<h4>`**…**`</h4>`**)
`h5` element (**`<h5>`**…**`</h5>`**)
`h6` element (**`<h6>`**…**`</h6>`**)
`head` element (**`<head>`**…**`</head>`**)
height
horizontal rule
`href` attribute of <**`a`**> element
HTML (HyperText Markup Language)
HTML document
`html` element (**`<html>`**…**`</html>`**)
HTML file
HTML tags
HTML-kit
hyperlink
hypertext
image
`img` element
`index.html`
line-break element (**`<br>`**…**`</br>`**)
link
`link` attribute of **`body`** element…
`mailto:`
markup language
opening tag
`p` element (paragraph; **`<p>`**…**`</p>`**)
presentation of a Web page
RGB colors
`size =` in **`<font>`**
source-code form
special characters
`src` attribute in **`img`** element
`strong` element (**`<strong>`**…**`</strong>`**)
structure of a Web page
`sub` (subscript)
`sup` (superscript)
tags in HTML
`text` in **`body`**
text-based browser
`title` element (**`<title>`**…**`</title>`**)
unordered list (**`<ul>`**…**`</ul>`**)
Web site
`width` attribute
width by percentage
width by pixel
World Wide Web

SELF-REVIEW EXERCISES

I.1 State whether the following statements are *true* or *false*. If *false*, explain why.

a) The document type for an HTML document is optional.
b) The use of the `em` and `strong` elements is deprecated.
c) The name of your site's home page should always be `homepage.html`.
d) It is a good programming practice to insert comments into your HTML document that explain what you are doing.
e) A hyperlink is inserted around text with the `link` element.

I.2 Fill in the blanks in each of the following statements:

a) The __________ element is used to insert a horizontal rule.
b) Superscript is formatted with the __________ element and subscript is formatted with the __________ element.
c) The __________ element is located within the `<head>`...`</head>` tags.
d) The least important header is the __________ element and the most important text header is __________.
e) The __________ element is used to create an unordered list.

I.3 Identify each of the following as either an element or attribute:

a) `html`
b) `width`
c) `href`
d) `br`
e) `h3`
f) `a`
g) `src`

ANSWERS TO SELF-REVIEW EXERCISES

I.1 a) False. The document type is required for HTMl documents. b) False. The use of the `i` and `b` elements is deprecated. Elements `em` and `strong` may be used instead. c) False. The name of your home page should always be `index.html`. d) True. e) False. A hyperlink is inserted around text with the `a` (anchor) element.

I.2 a) `hr`. b) `sup`, `sub`. c) `title`. d) `h6`, `h1`. e) `ul`.

I.3 a) Tag. b) Attribute. c) Attribute. d) Tag. e) Tag. f) Tag. g) Attribute.

EXERCISES

I.4 Use HTML to mark up the first paragraph of this appendix. Use `h1` for the section header, `p` for text, `strong` for the first word of every sentence, and `em` for all capital letters.

I.5 Why is this code valid? (*Hint*: you can find the W3C specification for the `p` element at `www.w3.org/TR/html4`)

```
<p>Here's some text...
<hr>
<p>And some more text...</p>
```

I.6 Why is this code invalid? [*Hint*: you can find the W3C specification for the `br` element at the same URL given in Exercise 2.5.]

```
<p>Here's some text...<br></br>
And some more text...</p>
```

I.7 We have an image named **`deitel.gif`** that is 200 pixels wide and 150 pixels high. Use the **`width`** and **`height`** attributes of the **`img`** tag to a) increase image size by 100%; b) increase image size by 50%; c) change the width-to-height ratio to 2:1, keeping the width attained in a).

I.8 Create a link to each of the following: a) **`index.html`**, located in the **`files`** directory; b) **`index.html`**, located in the **`text`** subdirectory of the **`files`** directory; c) **`index.html`**, located in the **`other`** directory in your *parent directory* [*Hint*: **`..`** signifies parent directory.]; d) A link to the President of the United States' email address (**`president@whitehouse.gov`**); e) An **`FTP`** link to the file named **`README`** in the **`pub`** directory of **`ftp.cdrom.com`**. [*Hint*: remember to use **`ftp://`**.]

J

Introduction to HyperText Markup Language 4: Part 2

Objectives

- To be able to create tables with rows and columns of data.
- To be able to control the display and formatting of tables.
- To be able to create and use forms.
- To be able to create and use image maps to aid hyperlinking.
- To be able to make Web pages accessible to search engines.
- To be able to use the **`frameset`** element to create more interesting Web pages.

Yea, from the table of my memory
I'll wipe away all trivial fond records.
William Shakespeare

Outline

- **I.1 Introduction**
- **I.2 Basic HTML Tables**
- **I.3 Intermediate HTML Tables and Formatting**
- **I.4 Basic HTML Forms**
- **I.5 More Complex HTML Forms**
- **I.6 Internal Linking**
- **I.7 Creating and Using Image Maps**
- **I.8 `<meta>` Tags**
- **I.9 `frameset` Element**
- **I.10 Nested `framesets`**
- **I.11 Internet and World Wide Web Resources**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

J.1 Introduction

In Appendix I, Introduction to HyperText Markup Language 4: Part 1, we discussed some basic HTML features. We built several complete Web pages featuring text, hyperlinks, images and such formatting tools as horizontal rules and line breaks.

In this appendix, we discuss more substantial HTML elements and features. We will see how to present information in *tables*. We discuss how to use forms to collect information from people browsing a site. We explain how to use *internal linking* and *image maps* to make pages more navigable. We also discuss how to use *frames* to make navigating Web sites easier. By the end of this appendix, you will be familiar with most commonly used HTML tags and features. You will then be able to create more complex Web sites. In this appendix, we do not present any C# programming.

J.2 Basic HTML Tables

HTML 4.0 *tables* are used to mark up tabular data, such as data stored in a database. The table in Fig. J.1 organizes data into rows and columns.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. J.1: table.html -->
<!-- Basic table design.  -->

<head>
   <title>C# How to Program - Tables</title>
</head>
```

Fig. J.1 HTML table. (Part 1 of 2.)

```
11
12  <body>
13
14     <h1>Table Example Page</h1>
15
16     <!-- the <table> tag opens a new table and lets you -->
17     <!-- put in design options and instructions         -->
18     <table border = "1" width = "40%">
19
20        <!-- use the <caption> tag to summarize the table's -->
21        <!-- contents (this helps the visually impaired)    -->
22        <caption>Here is a small sample table.</caption>
23
24        <!-- The <thead> is the first (non-scrolling)  -->
25        <!-- horizontal section. <th> inserts a header -->
26        <!--  cell and displays bold text              -->
27        <thead>
28           <tr><th>This is the head.</th></tr>
29        </thead>
30
31        <!-- All of your important content goes in the <tbody>. -->
32        <!-- Use this tag to format the entire section          -->
33        <!-- <td> inserts a data cell, with regular text        -->
34        <tbody>
35           <tr><td>This is the body.</td></tr>
36        </tbody>
37
38     </table>
39
40  </body>
41  </html>
```

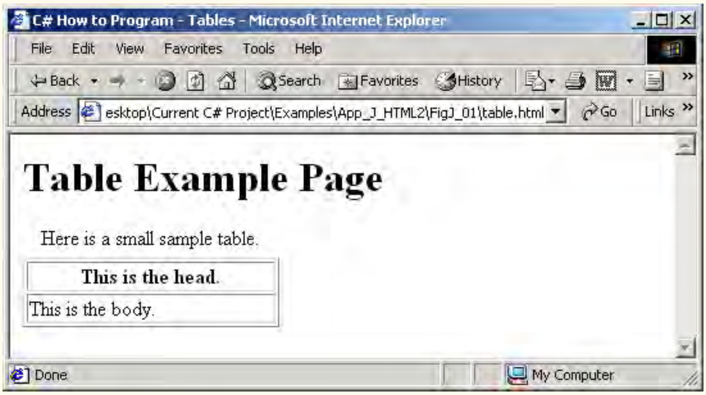

C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address esktop\Current C# Project\Examples\App_J_HTML2\FigJ_01\table.html
Go Links
Done
My Computer

Table Example Page

Here is a small sample table.

This is the head.
This is the body.

Fig. J.1 HTML table. (Part 2 of 2.)

All tags and text that apply to the table go inside the `<table>` element, which begins on line 18:

```
<table border = "1" width = "40%">
```

The `border` *attribute* lets you set the width of the table’s border in pixels. If you want the border to be invisible, you can specify `border = "0"`. In the table shown in Fig. J.1, the value of the border attribute is set to `1`. The `width` attribute sets the width of the table as either a number of pixels or a percentage of the screen width.

Line 22

```
<caption>Here is a small sample table.</caption>
```

inserts a `caption` element into the table. The text inside the `caption` element is inserted directly above the table in most visual browsers. The caption text is also used to help *text-based browsers* interpret the table data.

Tables can be split into distinct horizontal and vertical sections. The first of these sections, the head area, appears in lines 27–29

```
<thead>
   <tr><th>This is the head.</th></tr>
</thead>
```

Put all header information (for example, the titles of the table and column headers) inside the `thead` element. The `tr`, or *table row element*, is used to create rows of table cells. All of the cells in a row belong in the `<tr>` element for that row.

The smallest unit of the table is the *data cell*. There are two types of data cells, one type—the `th` element—is located in the table header. The other type—the `td` element—is located in the table body. The code example in Fig. J.1 inserts a header cell, using the `th` element. Header cells, which are placed in the `<thead>` element, are suitable for column headings.

The second grouping section, the `tbody` element, appears in lines 34–36

```
<tbody>
   <tr><td>This is the body.</td></tr>
</tbody>
```

Like `thead`, the `tbody` element is used for formatting and grouping purposes. Although there is only one row and one cell (line 35) in the above example, most tables will use `tbody` to group the majority of their content in multiple rows and multiple cells.

Look-and-Feel Observation J.1

Use tables in your HTML pages to mark up tabular data.

Common Programming Error J.1

Forgetting to close any of the elements inside the `table` *element is an error and can distort the table format. Be sure to check that every element is opened and closed in its proper place to make sure that the table is structured as intended.*

J.3 Intermediate HTML Tables and Formatting

In the previous section and code example, we explored the structure of a basic table. In Fig. J.2, we extend our table example with more structural elements and attributes

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. J.2: table.html       -->
<!-- Intermediate table design. -->

<head>
   <title>C# How to Program - Tables</title>
</head>

<body>

   <h1>Table Example Page</h1>

   <table border = "1">
      <caption>Here is a more complex sample table.</caption>

      <!-- <colgroup> and <col> are used to format     -->
      <!-- entire columns at once. SPAN determines how -->
      <!-- many columns the <col> tag effects.         -->
      <colgroup>
         <col align = "right">
         <col span = "4">
      </colgroup>

      <thead>

         <!-- rowspans and colspans combine the indicated -->
         <!-- number of cells vertically or horizontally  -->
         <tr>
            <th rowspan = "2">
               <img src = "camel.gif" width = "205"
                  height = "167" alt = "Picture of a camel">
            </th>
            <th colspan = "4" valign = "top">
               <h1>Camelid comparison</h1><br>
               <p>Approximate as of 8/99</p>
            </th>
         </tr>

         <tr valign = "bottom">
            <th># of Humps</th>
            <th>Indigenous region</th>
            <th>Spits?</th>
            <th>Produces Wool?</th>
         </tr>

      </thead>

```

Fig. J.2 Complex HTML table. (Part 1 of 2.)

```
51          <tbody>
52
53             <tr>
54                <th>Camels (bactrian)</th>
55                <td>2</td>
56                <td>Africa/Asia</td>
57                <td rowspan = "2">Llama</td>
58                <td rowspan = "2">Llama</td>
59             </tr>
60
61             <tr>
62                <th>Llamas</th>
63                <td>1</td>
64                <td>Andes Mountains</td>
65             </tr>
66
67          </tbody>
68
69       </table>
70
71    </body>
72    </html>
```

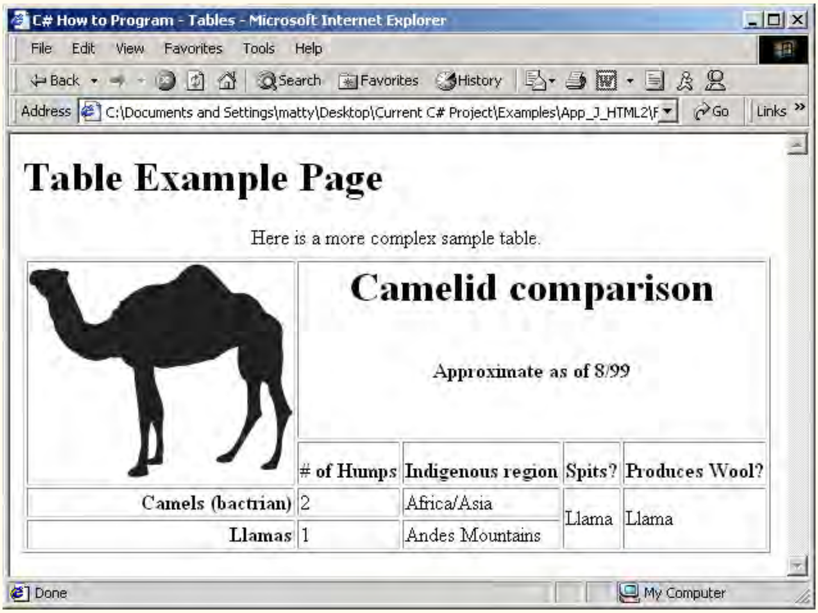

C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_J_HTML2\F
Go
Links
Done
My Computer

Table Example Page

Here is a more complex sample table.

Camelid comparison

Approximate as of 8/99

	# of Humps	Indigenous region	Spits?	Produces Wool?
Camels (bactrian)	2	Africa/Asia	Llama	Llama
Llamas	1	Andes Mountains		

Fig. J.2 Complex HTML table. (Part 2 of 2.)

The table begins on line 16. The `colgroup` *element*, used for grouping columns, is shown on lines 22–25

```
<colgroup>
   <col align = "right">
```

```
      <col span = "4">
   </colgroup>
```

The **`colgroup`** element can be used to group and format columns. Each ***`col`*** element in the **`<colgroup>`**…**`</colgroup>`** tags can format any number of columns (specified with the **`span`** attribute). Any formatting to be applied to a column or group of columns can be specified in both the **`colgroup`** and **`col`** tags. In this case, we align the text inside the leftmost column to the right. Another useful attribute to use here is **`width`**, which specifies the width of the column.

Most visual Web browsers automatically format data cells to fit the data they contain. However, it is possible to make some data cells larger than others. This effect is accomplished with the ***`rowspan`*** and ***`colspan`*** attributes, which can be placed inside any data cell element. The value of the attribute specifies the number of rows or columns to be occupied by the cell, respectively. For example, **`rowspan = "2"`** tells the browser that this data cell will span the area of two vertically adjacent cells. These cells will be joined vertically (and will thus span over two rows). An example of **`colspan`** appears in line 36,

```
   <th colspan = "4" valign = "top">
```

where the header cell is widened to span four cells.

We also see here an example of vertical alignment formatting. The ***`valign`*** attribute accepts the following values: **`"top"`**, **`"middle"`**, **`"bottom"`** and **`"baseline"`**. All cells in a row whose **`valign`** attribute is set to **`"baseline"`** will have the first text line occur on a common baseline. The default vertical alignment in all data and header cells is **`valign = "middle"`**.

The remaining code in Fig. J.2 demonstrates other uses of the **`table`** attributes and elements outlined above.

Common Programming Error J.2

*When using **`colspan`** and **`rowspan`** in table data cells, consider that the modified cells will cover the areas of other cells. Compensate for this in your code by reducing the number of cells in that row or column. If you do not, the formatting of your table will be distorted, and you could inadvertently create more columns and/or rows than you originally intended.*

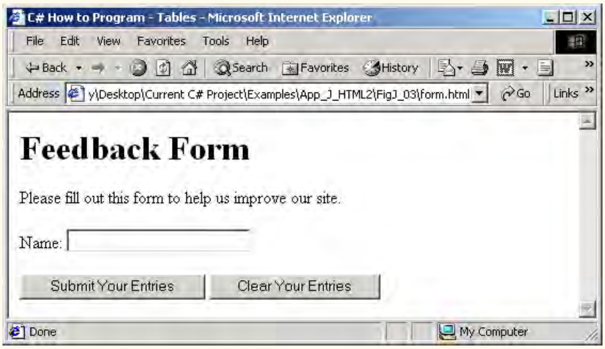
J.4 Basic HTML Forms

HTML provides several mechanisms to collect information from people viewing your site; one is the *form* (Fig. J.3).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. J.3: form.html    -->
<!-- Form design example 1. -->

<head>
   <title>C# How to Program - Tables</title>
</head>
```

Fig. J.3 Simple form with hidden fields and a text box. (Part 1 of 2.)

```

<body>

   <h1>Feedback Form</h1>

   <p>Please fill out this form to help us improve our site.</p>

   <!-- This tag starts the form, gives the method of sending -->
   <!-- information and the location of form scripts.        -->
   <!-- Hidden inputs give the server non-visual information  -->
   <form method = "post" action = "/cgi-bin/formmail">

   <p>
      <input type = "hidden" name = "recipient"
         value = "deitel@deitel.com">

      <input type = "hidden" name = "subject"
         value = "Feedback Form">

      <input type = "hidden" name = "redirect"
         value = "main.html">
   </p>

   <!-- <input type = "text"> inserts a text box -->
   <p><label>Name:
      <input name = "name" type = "text" size = "25">
   </label></p>

   <p>
      <!-- input types "submit" and "reset" insert buttons -->
      <!-- for submitting or clearing the form's contents  -->
      <input type = "submit" value = "Submit Your Entries">
      <input type = "reset" value = "Clear Your Entries">
   </p>

   </form>

</body>
</html>
```



C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address y\Desktop\Current C# Project\Examples\App_J_HTML2\FigJ_03\form.html
Go Links
Submit Your Entries
Clear Your Entries
Done
My Computer

Feedback Form

Please fill out this form to help us improve our site.

Name:

Fig. J.3 Simple form with hidden fields and a text box. (Part 2 of 2.)

The form begins on line 21

```
<form method = "post" action = "/cgi-bin/formmail">
```

with the ***`form`*** element. The ***`method`*** attribute indicates the way the information gathered in the form will be sent to the *Web server* for processing. Use ***`method = "post"`*** in a form that causes changes to server data, for example when updating a database. The form data will be sent to the server as an *environment variable*, which scripts are able to access. The other possible value, **`method = "get"`**, should be used when your form does not cause any changes in server-side data, for example when making a database request. The form data from **`method = "get"`** is appended to the end of the URL (for example, **`/cgi-bin/formmail?name=bob&order=5`**). Also be aware that **`method = "get"`** is limited to standard characters and cannot submit any special characters.

A *Web server* is a machine that runs a software package like Microsoft's PWS (Personal Web Server), Microsoft's IIS (Internet Information Server) or Apache. Web servers handle browser requests. When a browser requests a page or file somewhere on a server, the server processes the request and returns an answer to the browser. In this example, the data from the form goes to a CGI (Common Gateway Interface) script, which is a means of interfacing an HTML page with a script (i.e., a program) written in Perl, C, Tcl or other languages. The script then handles the data fed to it by the server and typically returns some information for the user. The **`action`** attribute in the **`form`** tag is the URL for this script; in this case, it is a simple script that emails form data to an address. Most Internet Service Providers (ISPs) will have a script like this on their site, so you can ask your system administrator how to set up your HTML to use the script correctly.

For this particular script, there are several pieces of information (not seen by the user) needed in the form. Lines 24–31

```
<input type = "hidden" name = "recipient"
   value = "deitel@deitel.com">

<input type = "hidden" name = "subject"
   value = "Feedback Form">

<input type = "hidden" name = "redirect"
   value = "main.html">
```

specify this information using *hidden input elements*. The **`input`** element is common in forms and always requires the ***`type`*** attribute. Two other attributes are ***`name`***, which provides a unique identifier for the **`input`** element, and ***`value`***, which indicates the value that the ***`input`*** element sends to the server upon submission.

As shown above, hidden inputs always have the attribute ***`type = "hidden"`***. The three hidden inputs shown are typical for this kind of CGI script: An email address to which the data will be sent, the subject line of the email and a URL to which the user is redirected after submitting the form.

Good Programming Practice J.1

Place hidden **`input`** *elements in the beginning of a form, right after the opening* **`<form>`** *tag. This makes these elements easier to find and identify.*

The usage of an **`input`** element is defined by the value of its **`type`** attribute. We introduce another of these options in lines 35–37:

```
<p><label>Name:
   <input name = "name" type = "text" size = "25">
</label></p>
```

The input ***`type = "text"`*** inserts a one-line text box into the form (line 36). A good use of the textual input element is for names or other one-line pieces of information. The **`label`** element on lines 35–37 provide a description for the **`input`** element on line 36.

We also use the ***`size`*** attribute of the **`input`** element to specify the width of the text input, measured in characters. You can also set a maximum number of characters that the text input will accept using the **`maxlength`** attribute.

Good Programming Practice J.2

*When using **`input`** elements in forms, be sure to leave enough space with the **`maxlength`** attribute for users to input the pertinent information.*

Common Programming Error J.3

*Forgetting to include a **`label`** element for each form element is a design error. Without these labels, users will have no way of knowing what the function of individual form elements is.*

There are two types of **`input`** elements in lines 42–43

```
<input type = "submit" value = "Submit Your Entries">
<input type = "reset" value = "Clear Your Entries">
```

that should be inserted into every form. The ***`type = "submit"`*** **`input`** element allows the user to submit the data entered in the form to the server for processing. Most visual Web browsers place a button in the form that submits the data when clicked. The **`value`** attribute changes the text displayed on the button (the default value is ***`"submit"`***). The input element ***`type = "reset"`*** allows a user to reset all form elements to the default values. This can help the user correct mistakes or simply start over. As with the **`submit`** input, the **`value`** attribute of the **`reset`** **`input`** element affects the text of the button on the screen, but does not affect its functionality.

Common Programming Error J.4

*Be sure to close your form code with the **`</form>`** tag. Neglecting to do so is an error and can affect the functionality of other forms on the same page.*

J.5 More Complex HTML Forms

We introduce additional form input options in Fig. J.4.

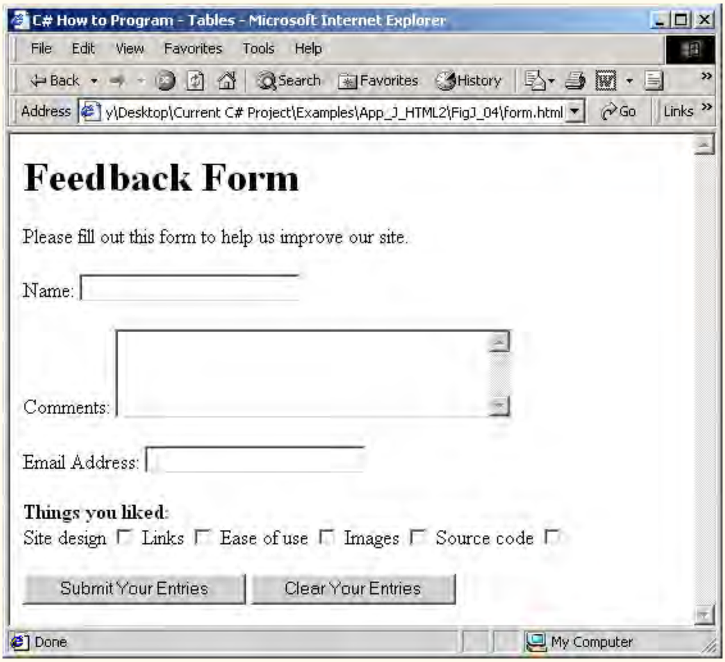
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. J.4: form.html    -->
<!-- Form design example 2. -->

<head>
   <title>C# How to Program - Tables</title>
</head>
```

Fig. J.4 Form including textareas, password boxes and checkboxes. (Part 1 of 3.)

```

<body>

   <h1>Feedback Form</h1>

   <p>Please fill out this form to help us improve our site.</p>

   <form method = "post" action = "/cgi-bin/formmail">

      <p>
         <input type = "hidden" name = "recipient"
            value = "deitel@deitel.com">

         <input type = "hidden" name = "subject"
            value = "Feedback Form">

         <input type = "hidden" name = "redirect"
            value = "main.html">
      </p>

      <p><label>Name:
         <input name = "name" type = "text" size = "25">
      </label></p>

      <!-- <textarea> creates a textbox of the size given -->
      <p><label>Comments:
         <textarea name = "comments" rows = "4" cols = "36">
         </textarea>
      </label></p>

      <!-- <input type = "password"> inserts textbox whose -->
      <!-- readout will be in *** not regular characters   -->
      <p><label>Email Address:
         <input name = "email" type = "password" size = "25">
      </label></p>

      <p>
         <strong>Things you liked:</strong><br>

         <label>Site design
         <input name = "thingsliked" type = "checkbox"
            value = "Design"></label>

         <label>Links
         <input name = "thingsliked" type = "checkbox"
            value = "Links"></label>

         <label>Ease of use
         <input name = "thingsliked" type = "checkbox"
            value = "Ease"></label>

         <label>Images
```

Fig. J.4 Form including textareas, password boxes and checkboxes. (Part 2 of 3.)

```
63                <input name = "thingsliked" type = "checkbox"
64                   value = "Images"></label>
65
66                <label>Source code
67                <input name = "thingsliked" type = "checkbox"
68                   value = "Code"></label>
69             </p>
70
71             <p>
72                <input type = "submit" value = "Submit Your Entries">
73                <input type = "reset" value = "Clear Your Entries">
74             </p>
75
76          </form>
77
78       </body>
79    </html>
```


C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address y\Desktop\Current C# Project\Examples\App_J_HTML2\FigJ_04\form.html
Go Links
Done
My Computer

Feedback Form

Please fill out this form to help us improve our site.

Name:

Comments:

Email Address:

Things you liked:

Site design ☐ Links ☐ Ease of use ☐ Images ☐ Source code ☐

Submit Your Entries | Clear Your Entries

Fig. J.4 Form including textareas, password boxes and checkboxes. (Part 3 of 3.)

Lines 37–38

```
<textarea name = "comments" rows = "4" cols = "36">
</textarea>
```

introduce the ***`textarea`*** element. The **`textarea`** element inserts a text box into the form. You specify the size of the box with the ***`rows`*** *attribute*, which sets the number of rows that will appear in the **`textarea`**. With the ***`cols`*** *attribute*, you specify how wide the `textarea` should be. This `textarea` is four rows of characters tall and 36 characters wide. Any default text that you want to place inside the `textarea` should be contained in the `textarea` element.

The input *`type = "password"`* (line 44)

```
<input name = "email" type = "password" size = "25">
```

inserts a text box with the indicated size. The password input field provides a way for users to enter information that the user would not want others to be able to read on the screen. In visual browsers, the data the user types into a password input field is shown as asterisks. However, the actual value the user enters is sent to the server. Nonvisual browsers may render this type of input field differently.

Lines 50–68 introduce another type of form element, the checkbox. Every `input` element with `type = "checkbox"` creates a new checkbox item in the form. Checkboxes can be used individually or in groups. Each checkbox in a group should have the same `name` (in this case, `name = "thingsliked"`). This notifies the script handling the form that all of the checkboxes are related to one another.

Common Programming Error J.5

When your form has several checkboxes with the same `name`, you must make sure that they have different `value`s, or else the script will have no way of distinguishing between them.

Additional form elements are introduced in Fig. J.5. In this form example, we introduce two new types of input options. The first of these is the *radio button*, introduced in lines 80–97. Inserted into forms with the `input` attribute *`type = "radio"`*, radio buttons are similar in function and usage to checkboxes. Radio buttons are different in that only one element in the group may be selected at any time. All of the `name` attributes of a group of radio inputs must be the same and all of the `value` attributes different. Insert the attribute *`checked`* to indicate which radio button you would like selected initially. The `checked` attribute can also be applied to checkboxes.

Common Programming Error J.6

When you are using a group of radio inputs in a form, forgetting to set the `name` values to the same name will let the user select all the radio buttons at the same time—an undesired result.

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
2            "http://www.w3.org/TR/html4/strict.dtd">
3  <html>
4
5  <!-- Fig. J.5: form.html    -->
6  <!-- Form design example 3. -->
7
8  <head>
9     <title>C# How to Program - Tables</title>
10 </head>
11
12 <body>
13
14    <h1>Feedback Form</h1>
15
```

Fig. J.5 Form including radio buttons and pulldown lists. (Part 1 of 4.)

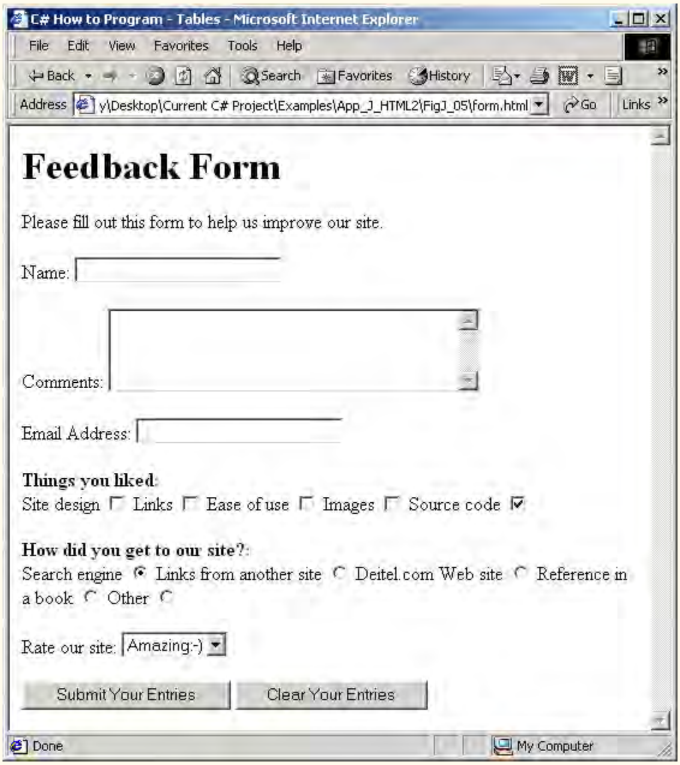
```
      <p>Please fill out this form to help us improve our site.</p>

      <form method = "post" action = "/cgi-bin/formmail">

         <p>
            <input type = "hidden" name = "recipient"
               value = "deitel@deitel.com">

            <input type = "hidden" name = "subject"
               value = "Feedback Form">

            <input type = "hidden" name = "redirect"
               value = "main.html">
         </p>

         <p><label>Name:
            <input name = "name" type = "text" size = "25">
         </label></p>

         <p><label>Comments:
            <textarea name = "comments" rows = "4" cols = "36">
               </textarea>
         </label></p>

         <p><label>Email Address:
            <input name = "email" type = "password" size = "25">
         </label></p>

         <p>
            <strong>Things you liked:</strong><br>

            <label>Site design
               <input name = "things" type = "checkbox"
                  value = "Design">
            </label>

            <label>Links
               <input name = "things" type = "checkbox"
                  value = "Links">
            </label>

            <label>Ease of use
               <input name = "things" type = "checkbox"
                  value = "Ease">
            </label>

            <label>Images
               <input name = "things" type = "checkbox"
                  value = "Images">
            </label>

            <label>Source code
```

Fig. J.5 Form including radio buttons and pulldown lists. (Part 2 of 4.)

```
            <input name = "things" type = "checkbox"
               value = "Code">
         </label>
      </p>

      <!-- <input type = "radio"> creates one radio button -->
      <!-- radio buttons and checkboxes differ in that      -->
      <!-- only one radio button in group can be selected   -->
      <p>
         <strong>How did you get to our site?:</strong><br>

         <label>Search engine
            <input name = "how get to site" type = "radio"
               value = "search engine" checked></label>

         <label>Links from another site
            <input name = "how get to site" type = "radio"
               value = "link"></label>

         <label>Deitel.com Web site
            <input name = "how get to site" type = "radio"
               value = "deitel.com"></label>

         <label>Reference in a book
            <input name = "how get to site" type = "radio"
               value = "book"></label>

         <label>Other
            <input name = "how get to site" type = "radio"
               value = "other"></label>

      </p>

      <!-- <select> tags present drop down menus   -->
      <!-- with choices indicated by <option> tags -->
      <p>
         <label>Rate our site:

         <select name = "rating">
            <option selected>Amazing:-)</option>
            <option>10</option>
            <option>9</option>
            <option>8</option>
            <option>7</option>
            <option>6</option>
            <option>5</option>
            <option>4</option>
            <option>3</option>
            <option>2</option>
            <option>1</option>
            <option>The Pits:-(</option>
         </select>

```

Fig. J.5 Form including radio buttons and pulldown lists. (Part 3 of 4.)

```
121             </label>
122          </p>
123 
124          <p>
125             <input type = "submit" value = "Submit Your Entries">
126             <input type = "reset" value = "Clear Your Entries">
127          </p>
128 
129       </form>
130 
131    </body>
132 </html>
```


C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address y\Desktop\Current C# Project\Examples\App_J_HTML2\FigJ_05\form.html Go Links

Feedback Form

Please fill out this form to help us improve our site.

Name:

Comments:

Email Address:

Things you liked:
Site design ☐ Links ☐ Ease of use ☐ Images ☐ Source code ☑

How did you get to our site?:
Search engine ● Links from another site ○ Deitel.com Web site ○ Reference in a book ○ Other ○

Rate our site: Amazing:-)

Submit Your Entries
Clear Your Entries
Done
My Computer

Fig. J.5 Form including radio buttons and pulldown lists. (Part 4 of 4.)

The last type of form input that we introduce here is the `select` element (lines 106–119). This will place a selectable list of items inside your form.

```
<select name = "rating">
   <option selected>Amazing:-)</option>
   <option>10</option>
   <option>9</option>
   <option>8</option>
   <option>7</option>
   <option>6</option>
```

```
      <option>5</option>
      <option>4</option>
      <option>3</option>
      <option>2</option>
      <option>1</option>
      <option>The Pits:-(</option>
   </select>
```

This type of form input is created via a **`select`** element. Inside the opening ***`<select>`*** tag, be sure to include the **`name`** attribute.

To add an item to the list, add to the **`select`** element an ***`option`*** element containing the text to be displayed. The **`selected`** attribute, like the **`checked`** attribute for radio buttons and checkboxes, applies a default selection to your list.

The preceding code will generate a pull-down list of options in most visual browsers, as shown in Fig. J.5. You can change the number of list options visible at one time, using the ***`size`*** attribute of the **`select`** element. Use this attribute if you prefer an expanded version of the list to the one-line expandable list.

J.6 Internal Linking

In Appendix I, Introduction to HyperText Markup Language 4: Part 1, we discussed how to link one Web page to another with text and image anchors. Figure J.6 introduces *internal linking*, which lets you create named anchors for hyperlinks to particular parts of an HTML document.

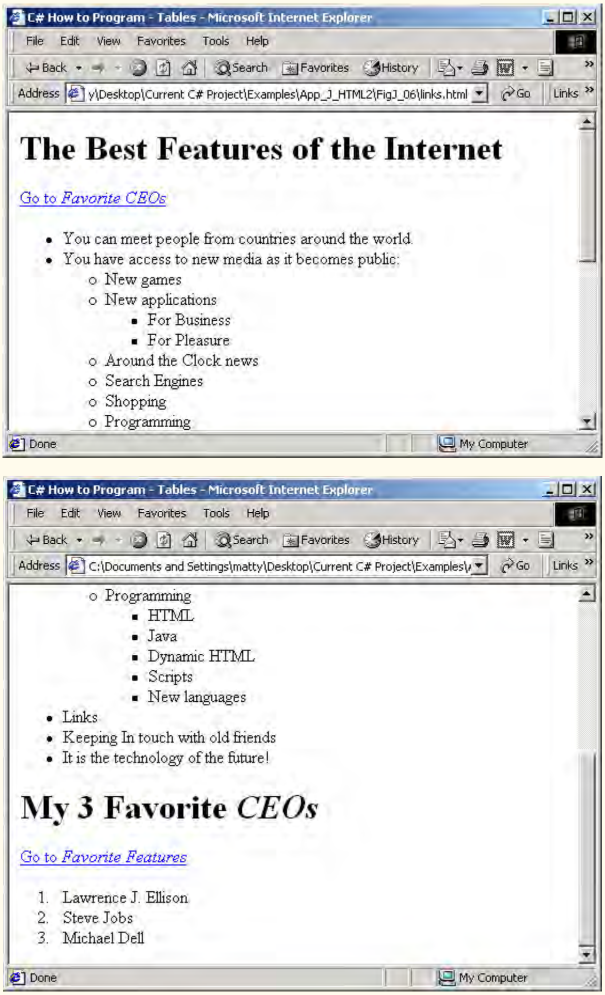
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. J.6: links.html -->
<!-- Internal linking.    -->

<head>
   <title>C# How to Program - Tables</title>
</head>

<body>

   <!-- <a name = ".."></a> makes internal hyperlinks -->
   <p>
      <a name = "features"></a>
   </p>

   <h1>The Best Features of the Internet</h1>

   <!-- internal link's address is "xx.html#linkname" -->
   <p>
      <a href = "#ceos">Go to <em>Favorite CEOs</em></a>
   </p>

```

Fig. J.6 Using internal hyperlinks to make your pages more navigable. (Part 1 of 3.)

```
      <ul>
         <li>You can meet people from countries around the world.
         </li>

         <li>You have access to new media as it becomes public:

            <ul>
               <li>New games</li>
               <li>New applications

                  <ul>
                     <li>For Business</li>
                     <li>For Pleasure</li>
                  </ul>

               </li>

               <li>Around the Clock news</li>
               <li>Search Engines</li>
               <li>Shopping</li>
               <li>Programming

                  <ul>
                     <li>HTML</li>
                     <li>Java</li>
                     <li>Dynamic HTML</li>
                     <li>Scripts</li>
                     <li>New languages</li>
                  </ul>

               </li>
            </ul>

         </li>

         <li>Links</li>
         <li>Keeping In touch with old friends</li>
         <li>It is the technology of the future!</li>
      </ul>

      <p><a name = "ceos"></a></p>

      <h1>My 3 Favorite <em>CEOs</em></h1>

      <p>
         <a href = "#features">Go to <em>Favorite Features</em></a>
      </p>

      <ol>
         <li>Lawrence J. Ellison</li>
         <li>Steve Jobs</li>
         <li>Michael Dell</li>
      </ol>
```

Fig. J.6 Using internal hyperlinks to make your pages more navigable. (Part 2 of 3.)

```
79
80   </body>
81   </html>
```


C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address y\Desktop\Current C# Project\Examples\App_J_HTML2\FigJ_06\links.html
Go Links
Done
My Computer

The Best Features of the Internet

Go to *Favorite CEOs*

- You can meet people from countries around the world.
- You have access to new media as it becomes public:
  - New games
  - New applications
    - For Business
    - For Pleasure
  - Around the Clock news
  - Search Engines
  - Shopping
  - Programming

C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\
Go Links
Done
My Computer

  - Programming
    - HTML
    - Java
    - Dynamic HTML
    - Scripts
    - New languages
- Links
- Keeping In touch with old friends
- It is the technology of the future!

My 3 Favorite *CEOs*

Go to *Favorite Features*

1. Lawrence J. Ellison
2. Steve Jobs
3. Michael Dell

Fig. J.6 Using internal hyperlinks to make your pages more navigable. (Part 3 of 3.)

Lines 15–17

```
<p>
   <a name = "features"></a>
</p>
```

show a named anchor for an internal hyperlink. A named anchor is created via an **`a`** element with a **`name`** attribute. Line 15 creates an anchor named **`features`**. Because the name of the page is **`list.html`**, the URL of this anchor in the Web page is **`list.html#features`**. Line 71

```
<a href = "#features">Go to <em>Favorite Features</em></a>
```

shows a hyperlink with the anchor **`features`** as its target. Selecting this hyperlink in a visual browser would scroll the browser window to the **`features`** anchor (line 16). Examples of this occur in Fig. J.6, which shows two different screen captures from the same page, each at a different anchor. You can also link to an anchor in another page, using the URL of that location (using the format **`href = "`*`page.html`*`#`*`name`*`"`**).

Look-and-Feel Observation J.2

Internal hyperlinks are most useful in large HTML files with lots of information. You can link to various points on the page to save the user from having to scroll down and find a specific location.

J.7 Creating and Using Image Maps

We have seen that images can be used as links to other places on your site or elsewhere on the Internet. We now discuss how to create *image maps* (Fig. J.7), which allow you to designate certain sections of the image as *hotspots* and then use these hotspots as links.

All elements of an image map are contained inside the ***`<map>`***…***`</map>`*** tags. The required attribute for the ***`map`*** element is **`name`** (line 18):

```
<map name = "picture">
```

As we will see, this attribute is needed for referencing purposes. A hotspot on the image is designated with the ***`area`*** element. Every ***`area`*** element has the following attributes: **`href`** sets the target for the link on that spot, ***`shape`*** and ***`coords`*** set the characteristics of the area and **`alt`** functions just as it does in the **`img`** element.

```
 1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 2            "http://www.w3.org/TR/html4/strict.dtd">
 3  <html>
 4
 5  <!-- Fig. J.7: picture.html         -->
 6  <!-- Creating and using imape maps. -->
 7
 8  <head>
 9     <title>C# How to Program - Tables</title>
10  </head>
11
12  <body>
13
14     <p>
15
16     <!-- <map> opens and names image map formatting -->
17     <!-- area and to be referenced later            -->
18     <map name = "picture">
19
20        <!-- "shape = rect" indicates rectangular -->
21        <!-- area, with coordinates of the        -->
22        <!-- upper-left and lower-right corners   -->
```

Fig. J.7 Picture with links anchored to an image map. (Part 1 of 2.)

```
      <area href = "form.html" shape = "rect"
         coords = "3, 122, 73, 143"
         alt = "Go to the feedback form">

      <area href = "contact.html" shape = "rect"
         coords = "109, 123, 199, 142"
         alt = "Go to the contact page">

      <area href = "main.html" shape = "rect"
         coords = "1, 2, 72, 17"
         alt = "Go to the homepage">

      <area href = "links.html" shape = "rect"
         coords = "155, 0, 199, 18"
         alt = "Go to the links page">

      <!-- "shape = polygon" indicates area of -->
      <!-- cusotmizable shape, with the        -->
      <!-- coordinates of every vertex listed  -->
      <area href = "mailto:deitel@deitel.com" shape = "poly"
      coords = "28, 22, 24, 68, 46, 114, 84, 111, 99, 56, 86, 13"
         alt = "Email the Deitels">

      <!-- "shape = circle" indicates circular -->
      <!-- area with center and radius listed  -->
      <area href = "mailto:deitel@deitel.com" shape = "circle"
         coords = "146, 66, 42" alt = "Email the Deitels">
   </map>

   <!-- <img src=... usemap = "#name"> says that    -->
   <!-- indicated image map will be used with image -->
   <img src = "deitel.gif" width = "200" height = "144"
      alt = "Harvey and Paul Deitel" usemap = "#picture">
   </p>

</body>
</html>
```

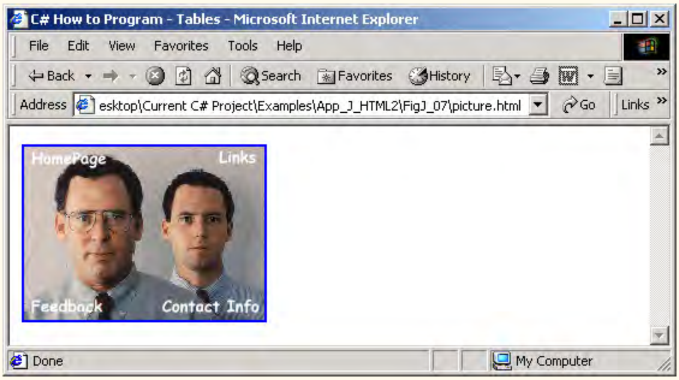

C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address esktop\Current C# Project\Examples\App_J_HTML2\FigJ_07\picture.html
Go Links
HomePage
Links
Feedback
Contact Info
Done
My Computer

Fig. J.7 Picture with links anchored to an image map. (Part 2 of 2.)

The markup on lines 23–25

```
<area href = "form.html" shape = "rect"
  coords = "3, 122, 73, 143" alt = "Go to the feedback form">
```

causes a *rectangular hotspot* to be drawn around the *coordinates* given in the **`coords`** element. A coordinate pair consists of two numbers, which are the locations of the point on the *x* and *y* axes. The *x* axis extends horizontally from the upper-left corner, the *y* axis vertically. Every point on an image has a unique *x*–*y* coordinate. In the case of a rectangular hotspot, the required coordinates are those of the upper-left and lower-right corners of the rectangle. In this case, the upper-left corner of the rectangle is located at 3 on the *x* axis and 122 on the *y* axis, annotated as (*3*, *122*). The lower-right corner of the rectangle is at (*73*, *143*).

Another map area is in lines 42–44

```
<area href = "mailto:deitel@deitel.com" shape = "poly"
   coords = "28, 22, 24, 68, 46, 114, 84, 111, 99, 56, 86, 13
   alt = "Email the Deitels">
```

In this case, we use the value ***`poly`*** for the **`shape`** attribute. This creates a hotspot in the shape of a polygon, using the coordinates in the **`coords`** attribute. These coordinates represent each vertex, or corner, of the polygon. The browser will automatically connect these points with lines to form the area of the hotspot.

`shape = "circle"` is the last shape attribute that is commonly used in image maps. It creates a *circular hotspot*, and requires both the coordinates of the center of the circle and the radius of the circle, in pixels.

To use the image map with an **`img`** element, you must insert the **`usemap = "#`***name***`"`** attribute into the **`img`** element, where *name* is the value of the **`name`** attribute in the **`map`** element. Lines 54–55

```
<img src = "deitel.gif" width = "200" height= "144" alt =
"Harvey and Paul Deitel" usemap = "#picture">
```

show how the image map **`name = "picture"`** is applied to the **`img`** element.

J.8 `<meta>` Tags

People use search engines to find interesting Web sites. Search engines usually catalog sites by following links from page to page and saving identification and classification information for each page visited. The main HTML element that search engines use to catalog pages is the ***`meta`*** tag (Fig. J.8).

A **`meta`** tag contains two attributes that should always be used. The first of these, ***`name`***, identifies the type of **`meta`** tag you are including. The ***`content`*** attribute provides information the search engine will catalog about your site.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>

<!-- Fig. J.8: main.html         -->
<!-- <meta> and <!doctype> tags. -->
```

Fig. J.8 Using **`meta`** to provide keywords and a description. (Part 1 of 2.)

```
 7
 8  <head>
 9     <!-- <meta> tags give search engines information -->
10     <!-- they need to catalog your site              -->
11     <meta name = "keywords" content = "Webpage, design, HTML,
12        tutorial, personal, help, index, form, contact, feedback,
13        list, links, frame, deitel">
14
15     <meta name = "description" content = "This Web site will help
16        you learn the basics of HTML and Webpage design through the
17        use of interactive examples and instruction.">
18
19     <title>C# How to Program - Tables</title>
20  </head>
21
22  <body>
23
24     <h1>Welcome to Our Web Site!</h1>
25
26     <p>
27        We have designed this site to teach about the wonders of
28        <em>HTML</em>. We have been using <em>HTML</em> since
29        version <strong>2.0</strong>, and we enjoy the features
30        that have been added recently. It seems only a short
31        time ago that we read our first <em>HTML</em> book.
32        Soon you will know about many of the great new
33        features of HTML 4.01.
34     </p>
35
36     <p>Have Fun With the Site!</p>
37
38  </body>
39  </html>
```

Fig. J.8 Using `meta` to provide keywords and a description. (Part 2 of 2.)

Lines 11–13 demonstrate the `meta` tag.

```
<meta name = "keywords" content = "Webpage, design, HTML,
   tutorial, personal, help, index, form, contact, feedback,
   list, links, frame, deitel">
```

The `content` of a `meta` tag with ***`name = "keywords"`*** provides search engines with a list of words that describe key aspects of your site. These words are used to match with searches—if someone searches for some of the terms in your `keywords meta` tag, they have a better chance of being notified about your site in the search-engine output. Thus, including `meta` tags and their `content` information will draw more viewers to your site.

The ***`description`*** attribute value (lines 15–17)

```
<meta name = "description" content = "This Web site will help
you learn the basics of HTML and Webpage design through the
use of interactive examples and instruction.">
```

is quite similar to the **`keywords`** value. Instead of giving a list of words describing your page, the **`content`**s of the keywords **`meta`** element should be a readable 3-to-4-line description of your site, written in sentence form. This description is also used by search engines to catalog and display your site.

Software Engineering Observation J.1

`meta` *elements are not visible to users of the site and must be placed inside the header section of your HTML document.*

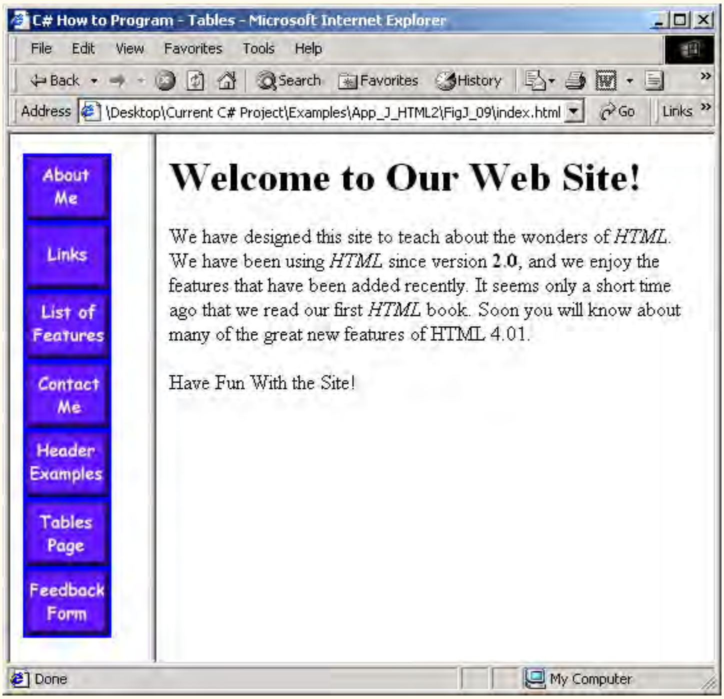
J.9 `frameset` Element

All of the Web pages we have designed so far have the ability to link to other pages but can display only one page at a time. Figure J.9 introduces *frames*, which can help you display more than one HTML file at a time. Frames, when used properly, can make your site more readable and usable for your users.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
          "http://www.w3.org/TR/html4/frameset.dtd">
<html>

<!-- Fig. J.9: index.html -->
<!-- HTML Frames I.       -->

<head>
   <meta name = "keywords" content = "Webpage, design, HTML,
      tutorial, personal, help, index, form, contact, feedback,
      list, links, frame, deitel">

   <meta name = "description" content = "This Web site will help
      you learn the basics of HTML and Webpage design through the
      use of interactive examples and instruction.">

   <title>C# How to Program - Tables</title>
</head>

<!-- the <frameset> tag gives dimensions of your frame -->
<frameset cols = "110,*">

   <!-- the individual frame elements specify -->
   <!-- which pages appear in given frames    -->
   <frame name = "nav" src = "nav.html">
   <frame name = "main" src = "main.html">

   <noframes>
      <p>
         This page uses frames, but your browser
            does not support them.
      </p>

```

Fig. J.9 Web site using two frames—navigation and content. (Part 1 of 2.)

```
34          <p>
35             Please, <a href = "nav.html">follow this link to
36                browse our site without frames</a>.
37          </p>
38       </noframes>
39
40    </frameset>
41    </html>
```


C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address \Desktop\Current C# Project\Examples\App_J_HTML2\FigJ_09\index.html
Go Links
About Me
Links
List of Features
Contact Me
Header Examples
Tables Page
Feedback Form
Done
My Computer

Welcome to Our Web Site!

We have designed this site to teach about the wonders of *HTML*. We have been using *HTML* since version **2.0**, and we enjoy the features that have been added recently. It seems only a short time ago that we read our first *HTML* book. Soon you will know about many of the great new features of HTML 4.01.

Have Fun With the Site!

Fig. J.9 Web site using two frames—navigation and content. (Part 2 of 2.)

On lines 1 and 2,

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
           "http://www.w3.org/TR/html4/frameset.dtd">
```

we encounter a new document type. The document type specified here indicates that this HTML document uses frames. You should use this document type whenever you use frames in your HTML document.

The framed page begins with the opening ***`frameset`*** tag, on line 21

```
<frameset cols = "110,*">
```

This tag tells the browser that the page contains frames. The **`cols`** attribute of the opening **`frameset`** tag gives the layout of the frameset. The value of ***`cols`*** (or ***`rows`***, if you will be writing a frameset with a horizontal layout) gives the width of each frame, either in pixels or as a percentage of the screen. In this case, the attribute `cols = "110,*"` tells the browser that there are two frames. The first one extends 110 pixels from the left edge of the screen, and the second frame fills the remainder of the screen (as indicated by the asterisk).

Now that we have defined the page layout, we have to specify what files will make up the frameset. We do this with the *`frame`* element in lines 25 and 26:

```
<frame name = "nav" src = "nav.html">
<frame name = "main" src = "main.html">
```

In each `frame` element, the `src` attribute gives the URL of the page that will be displayed in the frame. In the preceding example, the first frame (which covers 110 pixels on the left side of the `frameset`) will display the page `nav.html` and has the attribute `name = "nav"`. The second frame will display the page `main.html` and has the attribute `name = "main"`.

The purpose of a `name` attribute in the `frame` element is to identify the frame, enabling hyperlinks in a `frameset` to load in their intended target `frame`. For example,

```
<a href = "links.html" target = "main">
```

would load `links.html` in the frame whose `name` attribute is `"main"`.

A target in an anchor element can also be set to a number of preset values: `target="_blank"` loads the page in a new blank browser window, `target="_self"` loads the page into the same window as the anchor element, `target="_parent"` loads it in the parent `frameset` (i.e., the `frameset` which contains the current frame) and `target="_top"` loads the page into the full browser window (the page loads over the `frameset`).

In lines 28–38 of the code example in Fig. J.9, the `noframes` element displays HTML in those browsers that do not support frames.

Portability Tip J.1

Not everyone uses a browser that supports frames. Use the `noframes` *element inside the* `frameset` *to direct users to a nonframed version of your site.*

Look-and-Feel Observation J.3

Frames are capable of enhancing your page, but are often misused. Never use frames to accomplish what you could with other, simpler HTML formatting.

J.10 Nested framesets

You can use the `frameset` element to create more complex layouts in a framed Web site by nesting `frameset` areas as in Fig. J.10.

The first level of `frameset` tags is on lines 21 and 22

```
<frameset cols = "110,*">
   <frame name = "nav"src = "nav.html">
```

The `frameset` and `frame` elements here are constructed in the same manner as in Fig. J.9. We have one frame that extends over the first 110 pixels, starting at the left edge.

The second (nested) level of the `frameset` element covers only the remaining `frame` area that was not included in the primary `frameset`. Thus, any frames included in the second **`frameset`** will not include the leftmost 110 pixels of the screen. Lines 26–29 show the second level of **`frameset`** tags.

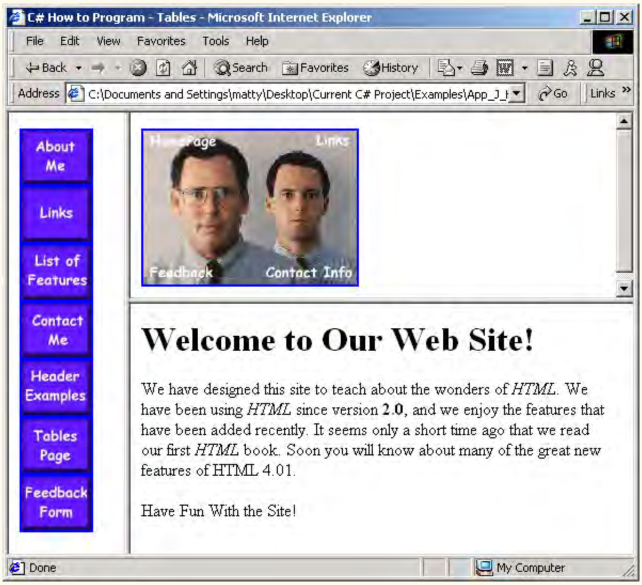
```
<frameset rows = "175,*">
   <frame name = "picture" src = "picture.html">
   <frame name = "main" src = "main.html">
</frameset>
```

In this **`frameset`** area, the first frame extends 175 pixels from the top of the screen, as indicated by the **`rows = "175,*"`**. Be sure to include the correct number of **`frame`** elements inside the second **`frameset`** area. Also, be sure to include a **`noframes`** element and to close both of the **`frameset`** areas at the end of the Web page.

```
 1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
 2            "http://www.w3.org/TR/html4/frameset.dtd">
 3  <html>
 4
 5  <!-- Fig. J.10: index.html  -->
 6  <!-- HTML frames II.        -->
 7
 8  <head>
 9
10     <meta name = "keywords" content = "Webpage, design, HTML,
11        tutorial, personal, help, index, form, contact, feedback,
12        list, links, frame, deitel">
13
14     <meta name = "description" content = "This Web site will help
15        you learn the basics of HTML and Webpage design through
16        the use of interactive examples and instruction.">
17
18     <title>C# How to Program - Tables</title>
19  </head>
20
21  <frameset cols = "110,*">
22     <frame name = "nav" src = "nav.html">
23
24     <!-- nested framesets are used to change formatting -->
25     <!-- and spacing of frameset as whole               -->
26     <frameset rows = "175,*">
27        <frame name = "picture" src = "picture.html">
28        <frame name = "main" src = "main.html">
29     </frameset>
30
31     <noframes>
32        <p>
33           This page uses frames, but your browser does
34              not support them.
35        </p>
36
37        <p>
38           Please, <a href = "nav.html">follow this link
39              to browse our site without frames</a>.
```

Fig. J.10 Framed Web site with a nested frameset. (Part 1 of 2.)

```
      </p>
   </noframes>

</frameset>
</html>
```


C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_J_I
Go Links
About Me
Links
List of Features
Contact Me
Header Examples
Tables Page
Feedback Form
HomePage
Links
Feedback
Contact Info
Done
My Computer

Welcome to Our Web Site!

We have designed this site to teach about the wonders of *HTML*. We have been using *HTML* since version **2.0**, and we enjoy the features that have been added recently. It seems only a short time ago that we read our first *HTML* book. Soon you will know about many of the great new features of HTML 4.01.

Have Fun With the Site!

Fig. J.10 Framed Web site with a nested frameset. (Part 2 of 2.)

Testing and Debugging Tip J.1

When using nested **`frameset`** *elements, indent every level of* **`frame`** *tag. This makes the page clearer and easier to debug.*

Look-and-Feel Observation J.4

Nested **`frameset`***s can help you create visually pleasing, easy-to-navigate Web sites.*

J.11 Internet and World Wide Web Resources

There are many Web sites that cover the more advanced and difficult features of HTML. Several of these sites are featured here.

`www.geocities.com/SiliconValley/Orchard/5212`
Adam's Advanced HTML Page is geared to those looking to master the more advanced techniques of HTML. It includes instructions for creating tables, frames and marquees and other advanced topics.

`www.w3scripts.com/html`
This site, an offshoot of *W3Schools*, is a repository for code examples exhibiting all of the features of HTML, from beginner to advanced.

`www.blooberry.com/indexdot/html`
Index Dot HTML, The Advance HTML Reference... The name speaks for itself. This site has a great directory and tree-based index of all HTML elements, plus more.

`www.neiljohan.com/html/advancedhtml.htm`
The *Advanced HTML Guide* gives insights into improving your site using HTML in ways you might not have thought possible.

SUMMARY

- HTML tables organize data into rows and columns. All tags and text that apply to a table go inside the **`<table>`**…**`</table>`** tags. The **`border`** attribute lets you set the width of the table's border in pixels. The **`width`** attribute sets the width of the table—you specify either a number of pixels or a percentage of the screen width.
- The text inside the **`<caption>`**…**`</caption>`** tags is inserted directly above the table in the browser window. The caption text is also used to help text-based browsers interpret the table data.
- Tables can be split into distinct horizontal and vertical sections. Put all header information (such as table titles and column headers) inside the **`<thead>`**…**`</thead>`** tags. The **`tr`** (table row) element is used for formatting the cells of individual rows. All of the cells in a row belong within the **`<tr>`**…**`</tr>`** tags of that row.
- The smallest area of the table that we are able to format is the data cell. There are two types of data cells: ones located in the header (**`<th>`**…**`</th>`**) and ones located in the table body (**`<td>`**…**`</td>`**). Header cells, usually placed in the **`<thead>`** area, are suitable for titles and column headings.
- Like **`thead`**, the **`tbody`** is used for formatting and grouping purposes. Most tables use **`tbody`** to house the majority of their content.
- **`td`** table data cells are left aligned by default. **`th`** cells are centered by default.
- Just as you can use the **`thead`** and **`tbody`** elements to format groups of table rows, you can use the **`colgroup`** element to group and format columns. **`colgroup`** is used by setting in its opening tag the number of columns it affects and the formatting it imposes on that group of columns.
- Each **`col`** element contained inside the **`<colgroup>`**…**`</colgroup>`** tags can in turn format a specified number of columns.
- You can add a background color or image to any table row or cell with either the **`bgcolor`** or **`background`** attributes, which are used in the same way as in the **`body`** element.
- It is possible to make some table data cells larger than others by using the **`rowspan`** and **`colspan`** attributes. The attribute value extends the data cell to span the specified number of cells.
- The **`valign`** (vertical alignment) attribute of a table data cell accepts the following values: **`"top"`**, **`"middle"`**, **`"bottom"`** and **`"baseline"`**.
- All cells in a table row whose **`valign`** attribute is set to **`"baseline"`** will have the first text line on a common baseline.
- The default vertical alignment in all data and header cells is **`valign="middle"`**.
- HTML provides several mechanisms—including the **`form`**—to collect information from people viewing your site.
- Use **`method = "post"`** in a form that causes changes to server data, for example when updating a database. The form data will be sent to the server as an environment variable, which scripts are able to access. The other possible value, **`method = "get"`**, should be used when your form does not cause any changes in server-side data, for example when making a database request. The form data from **`method = "get"`** is appended to the end of the URL. Because of this, the amount of data submitted using this **`method`** is limited to 4K. Also be aware that **`method = "get"`** is limited to standard characters and cannot submit any special characters.

- A Web server is a machine that runs a software package like Apache or IIS; servers are designed to handle browser requests. When a user uses a browser to request a page or file somewhere on the server, the server processes this request and returns an answer to the browser.
- The `action` attribute in the `form` tag is the path to a script that processes the form data.
- The input element is common in forms and always requires the `type` attribute. Two other attributes are `name`, which provides a unique identification for the `input`, and `value`, which indicates the value that the `input` element sends to the server upon submission.
- The input `type="text"` inserts a one-line text bar into the form. The value of this `input` element and the information that the server sends to you from this `input` is the text that the user types into the bar. The `size` attribute determines the width of the text input, measured in characters. You can also set a maximum number of characters that the text input will accept by inserting the `maxlength="`*length*`"` attribute.
- You must make sure to include a `label` element for each form element to indicate the function of the element.
- The `type="submit"` `input` element places a button in the form that submits data to the server when clicked. The `value` attribute of the `submit` input changes the text displayed on the button.
- The `type="reset"` input element places a button on the form that, when clicked, will clear all entries the user has entered into the form.
- The `textarea` element inserts a box into the form. You specify the size of the box (which is scrollable) inside the opening `<textarea>` tag with the `rows` attribute and the `cols` attribute.
- Data entered in a `type="password"` input appears on the screen as asterisks. The password is used for submitting sensitive information that the user would not want others to be able to read. It is just the browser that displays asterisks—the real form data is still submitted to the server.
- Every `input` element with `type="checkbox"` creates a new checkbox in the form. Checkboxes can be used individually or in groups. Each checkbox in a group should have the same `name` (in this case, `name="things"`).
- Inserted into forms by means of the `input` attribute `type="radio"`, radio buttons are different from checkboxes in that only one in the group may be selected at any time. All of the `name` attributes of a group of radio inputs must be the same and all of the `value` attributes different.
- Insert the attribute `checked` to indicate which radio button you would like selected initially.
- The `select` element places a selectable list of items inside your form. To add an item to the list, insert an `option` element in the `<select>`…`</select>` area and type what you want the list item to display on the same line. You can change the number of list options visible at one time by including the `size="`*size*`"` attribute inside the `<select>` tag. Use this attribute if you prefer an expanded version of the list to the one-line expandable list.
- A location on a page is marked by including a `name` attribute in an `a` element. Clicking this hyperlink in a browser would scroll the browser window to that point on the page.
- An image map allows you to designate certain sections of the image as hotspots and then use these hotspots as anchors for linking.
- All elements of an image map are contained inside the `<map>`…`</map>` tags. The required attribute for the `map` element is `name`.
- A hotspot on the image is designated with the ***`area`*** element. Every ***`<area>`*** tag has the following attributes: `href` sets the target for the link on that spot, ***`shape`*** and ***`coords`*** set the characteristics of the area and `alt` function just as it does in `<img>` tags.
- `shape="rect"` creates a rectangular hotspot around the *coordinates* of a `coords` element.

- A coordinate pair consists of two numbers, which are the locations of the point on the *x* and *y* axes. The *x* axis extends horizontally from the upper-left corner, the *y* axis vertically. Every point on an image has a unique *x*–*y* coordinate, annotated as *(x, y)*.
- In the case of a rectangular hotspot, the required coordinates are those of the upper-left and lower-right corners of the rectangle.
- The `shape="poly"` creates a hotspot of no preset shape—you specify the shape of the hotspot in the `coords` attribute by listing the coordinates of every vertex, or corner of the hotspot.
- `shape="circle"` creates a circular hotspot; it requires both the coordinates of the center of the circle and the length of the radius, in pixels.
- To use an image map with a graphic on your page, you must insert the `usemap="#name"` attribute into the `img` element, where “name” is the value of the `name` attribute in the `map` element.
- The main element that interacts with search engines is the `meta` element.
- `meta` tags contain two attributes that should always be used. The first of these, `name`, is an identification of the type of `meta` tag you are including. The `content` attribute gives the information the search engine will be cataloging.
- The `content` of a `meta` tag with `name="keywords"` provides the search engines with a list of words that describe the key aspects of your site. By including `meta` tags and their content information, you can give precise information about your site to search engines. This will help you draw a more focused audience to your site.
- The `description` value of the `name` attribute in the `meta` tag should be a 3-to-4-line description of your site, written in sentence form. This description is used by the search engine to catalog and display your site.
- `meta` elements are not visible to users of the site and should be placed inside the header section of your HTML document.
- The `frameset` tag tells the browser that the page contains frames.
- `cols` or `rows` gives the width of each frame in pixels or as a percentage of the screen.
- In each `frame` element, the `src` attribute gives the URL of the page that will be displayed in the specified frame.
- The purpose of a `name` attribute in the `frame` element is to give an identity to that specific frame, in order to enable hyperlinks in a `frameset` to load their intended `frame`. The `target` attribute in an anchor element is set to the `name` of the `frame` in which the new page should load.
- A target in an anchor element can be set to a number of preset values: `target="_blank"` loads the page in a new blank browser window, `target="self"` loads the page into the same window as the anchor element, `target="_parent"` loads the page into the parent `frameset` and `target="_top"` loads the page into the full browser window.
- Not everyone viewing a page has a browser that can handle frames. You therefore need to include a `noframes` element inside of the `frameset`. You should include regular HTML tags and elements within the `<noframes>…</noframes>` tags. Use this area to direct the user to a non-framed version of the site.
- By nesting `frameset` elements, you can create more complex layouts.

TERMINOLOGY

`<!doctype…>`
`<meta>` tag
`<option>`
`ACTION` attribute in `form` element
`area`
`border` property of `table` element
`caption` element
cell of a table

CGI script
checked
circular hotspot
col element
colgroup element
cols attribute of **table** element
colspan attribute of **td** element
column of a table
coords attribute inside **area** element
data cell
environment variable
form
frame element (**<frame>**…**</frame>**)
frameset element
header cell
hotspot
image map
indenting lists
input element (**<input>**…**</input>**)
input type="button"
input type="checkbox"
input type="password"
input type="radio"
input type="reset"
input type="submit"
input type="text"
input type="textarea"
internal linking
list
map element
maxlength="#"
method="get"
method="post"
name attribute in **input** element
name="recipient" in **input** element
name="redirect" in **input** element
name="subject" in **input** element
nested lists
noframes
noresize attribute in **frame**
ol (ordered list) element (**
…
**)
rectangular hotspot
row of a table
rowspan attribute of **td** element
scrolling attribute in **frame**
select element (**<select>**…**</select>**)
shape attribute inside **area** element
size attribute in **select**
src attribute of **frame** element
table
table element (**<table>**…**</table>**)
target="_blank"
target="_blank"
target="_parent"
target="_top"
tbody
td (table data) element (**<td>**…**</td>**)
text-based browser
th (header cell) element (**<th>**…**</th>**)
thead element (**<thead>**…**</thead>**)
tr (table row) element (**<tr>**…**</tr>**)
type=1 attribute of **<ol>**
type=a attribute of **<ol>**
type=A attribute of **<ol>**
type=i attribute of **<ol>**
type=I attribute of **<ol>**
ul (unordered list) element (**
…
**)
usemap="name" attribute in **img**
value attribute of **input** element
Web server

SELF-REVIEW EXERCISES

J.1 State whether the following statements are *true* or *false*. If *false*, explain why.
a) The width of all data cells in a table must be the same.
b) The **thead** element is mandatory in a **table**.
c) You are limited to a maximum of 100 internal links per page.
d) All browsers can render **frameset**s.

J.2 Fill in the blanks in each of the following statements.
a) The __________ attribute in an **input** element inserts a button that, when clicked, will clear the contents of the form.
b) The spacing of a **frameset** is set by including the __________ attribute or the __________ attribute inside of the **<frameset>** tag.
c) The __________ element inserts a new item in a list.

d) The __________ element tells the browser what version of HTML is included on the page. Two types of this element are __________ and __________.
e) The common shapes used in image maps are __________, __________ and __________.

J.3 Write HTML tags to accomplish the following tasks:
a) Insert a framed Web page with the first frame extending 300 pixels across the page from the left side.
b) Insert an ordered list that will have numbering by lowercase Roman numerals.
c) Insert a scrollable list (in a form) that will always display four entries of the list.
d) Insert an image map onto a page, using **`deitel.gif`** as an image and **`map`** with **`name="hello"`** as the image map, and have "**`hello`**" be the **`alt`** text.

ANSWERS TO SELF-REVIEW EXERCISES

J.1 a) False. You can specify the width of any column either in pixels or as a percentage of the total width of the table. c) False. The **`thead`** element is used only for formatting purposes and is optional (but it is recommended that you include it). d) False. You can have an unlimited number of hyperlink locations on any page. e) False. Text-based browsers are unable to render a **`frameset`** and must therefore rely on the information that you include inside the **`<noframes>`**…**`</noframes>`** tag.

J.2 a) **`type = "reset"`**. b) **`cols`**, **`rows`**. c) **`li`**. d) **`<!doctype`**…**`>`**, **`transitional`**, **`frameset`**. e) **`poly`**, **`circle`**, **`rect`**.

J.3 a) **`<frameset cols = "300,*">`**…**`</frameset>`** b) **`<ol type = "i">`**…**`</ol>`** c) **`<select size = "4">`**…**`</select>`** d)**`<img src = "deitel.gif" alt = "hello" usemap = "#hello">`**

EXERCISES

J.4 Categorize each of the following as an element or an attribute:
a) **`width`**
b) **`td`**
c) **`th`**
d) **`frame`**
e) **`name`**
f) **`select`**
g) **`type`**

J.5 What will the **`frameset`** produced by the following code look like? Assume that the pages being imported are blank with white backgrounds and that the dimensions of the screen are 800 by 600. Sketch the layout, approximating the dimensions.

```
<frameset rows = "20%,*">
<frame src = "hello.html" name = "hello">
   <frameset cols = "150,*">
   <frame src = "nav.html" name = "nav">
   <frame src = "deitel.html" name = "deitel">
   </frameset>
</frameset>
```

J.6 Assume that you have a document with many subsections. Write the HTML markup to create a frame with a table of contents on the left side of the window, and have each entry in the table of contents use internal linking to scroll down the document frame to the appropriate subsection.

K

Introduction to XHTML: Part 1

Objectives

- To understand important components of XHTML documents.
- To use XHTML to create World Wide Web pages.
- To be able to add images to Web pages.
- To understand how to create and use hyperlinks to navigate Web pages.
- To be able to mark up lists of information.

To read between the lines was easier than to follow the text.
Aristophanes

Outline

K.1 Introduction
K.2 Editing XHTML
K.3 First XHTML Example
K.4 W3C XHTML Validation Service
K.5 Headers
K.6 Linking
K.7 Images
K.8 Special Characters and More Line Breaks
K.9 Unordered Lists
K.10 Nested and Ordered Lists
K.11 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

K.1 Introduction

In this appendix, we introduce *XHTML*[1]—the *Extensible HyperText Markup Language*. In the next appendix, Introduction to XHTML: Part 2, we introduce more sophisticated XHTML techniques, such as *tables,* which are particularly useful for structuring information from *databases* (i.e., software that stores structured sets of data). In this appendix, we do not present any C# programming.

Unlike procedural programming languages such as C, Fortran, Cobol and Visual Basic, XHTML is a *markup language* that specifies the format of text that is displayed in a Web browser such as Microsoft's Internet Explorer or Netscape's Communicator.

One key issue when using XHTML[2] is the separation of the *presentation of a document* (i.e., the document's appearance when rendered by a browser) from the *structure of the document's information*. Throughout this appendix and the next, we will discuss this issue in depth.

K.2 Editing XHTML

In this appendix, we write XHTML in its *source-code form*. We create *XHTML documents* by typing them in with a text editor (e.g., *Notepad*, *Wordpad*, *vi* or *emacs*), saving the documents with either an ***`.html`*** or ***`.htm`*** file-name extension.

---

[1] XHTML has replaced the HyperText Markup Language (HTML) as the primary means of describing Web content. XHTML provides more robust, richer and more extensible features than HTML. For more on XHTML/HTML, visit **`www.w3.org/markup`**.

[2] As this book was being submitted to the publisher, XHTML 1.1 became a World Wide Web Consortium (W3C) Recommendation. The XHTML examples presented in this book are based upon the XHTML 1.0 Recommendation, because Internet Explorer 5.5 does not support the full set of XHTML 1.1 features. In the future, Internet Explorer and other browsers will support XHTML 1.1. When this occurs, we will update our Web site (**`www.deitel.com`**) with XHTML 1.1 examples and information.

Good Programming Practice K.1

Assign documents file names that describe their functionality. This practice can help you identify documents faster. It also helps people who want to link to a page, by giving them an easy-to-remember name. For example, if you are writing an XHTML document that contains product information, you might want to call it ***`products.html`****.*

Machines running specialized software called a *Web server* store XHTML documents. Clients (e.g., Web browsers) request specific *resources*, such as the XHTML documents from the Web server. For example, typing **`www.deitel.com/books/downloads.htm`** into a Web browser's address field requests **`downloads.htm`** from the Web server running at **`www.deitel.com`**. This document is located in a directory named **`books`**.

K.3 First XHTML Example

In this appendix and the next, we present XHTML markup and provide screen captures that show how Internet Explorer renders (i.e., displays) the XHTML. Every XHTML document we show has line numbers for the reader's convenience. These line numbers are not part of the XHTML documents.

Our first example (Fig. K.1) is an XHTML document named **`main.html`** that displays the message **`Welcome to XHTML!`** in the browser.

The key line in the program is line 14, which tells the browser to display **`Welcome to XHTML!`** Now let us consider each line of the program.

Lines 1–3 are required in XHTML documents to conform with proper XHTML syntax. For now, copy and paste these lines into each XHTML document you create. The meaning of these lines is discussed in detail in Chapter 18, Extensible Markup Language (XML).

Lines 5–6 are *XHTML comments*. XHTML document creators insert comments to improve markup readability and describe the content of a document. Comments also help other people read and understand an XHTML document's markup and content. Comments do not cause the browser to perform any action when the user loads the XHTML document into the Web browser to view the document. XHTML comments always start with **`<!--`** and end with **`-->`**. Each of our XHTML examples includes comments that specify the figure number and file name and provide a brief description of the example's purpose. Subsequent examples include comments in the markup, especially to highlight new features.

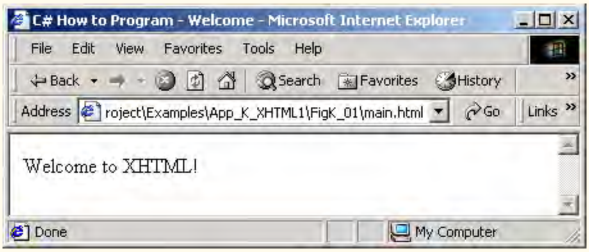
Good Programming Practice K.2

Place comments throughout your markup. Comments help other programmers understand the markup, assist in debugging and list useful information that you do not want the browser to render. Comments also help you understand your own markup when you revisit a document for modifications or updates in the future.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. K.1: main.html -->
6  <!-- Our first Web page. -->
7
```

Fig. K.1 First XHTML example. (Part 1 of 2.)

```
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>C# How to Program - Welcome</title>
11     </head>
12
13     <body>
14        <p>Welcome to XHTML!</p>
15     </body>
16  </html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address roject\Examples\App_K_XHTML1\FigK_01\main.html Go Links
Done
My Computer

Welcome to XHTML!

Fig. K.1 First XHTML example. (Part 2 of 2.)

XHTML markup contains text that represents the content of a document and *elements* that specify a document’s structure. Some important elements of an XHTML document include the ***`html`*** element, the ***`head`*** element and the ***`body`*** element. The **`html`** element encloses the *head section* (represented by the ***`head`*** *element*) and the *body section* (represented by the ***`body`*** *element*). The head section contains information about the XHTML document, such as the *title* of the document. The head section also can contain special document formatting instructions called *style sheets* and client-side programs called *scripts* for creating dynamic Web pages. The body section contains the page’s content that the browser displays when the user visits the Web page.

XHTML documents delimit an element with *start* and *end* tags. A start tag consists of the element name in angle brackets (e.g., **`<html>`**). An end tag consists of the element name preceded by a **`/`** in angle brackets (e.g., **`</html>`**). In this example, lines 8 and 16 define the start and end of the **`html`** element. Note that the end tag on line 16 has the same name as the start tag, but is preceded by a **`/`** inside the angle brackets. Many start tags define *attributes* that provide additional information about an element. Browsers can use this additional information to determine how to process the element. Each attribute has a *name* and a *value* separated by an equal sign (**`=`**). Line 8 specifies a required attribute (**`xmlns`**) and value (**`http://www.w3.org/1999/xhtml`**) for the **`html`** element in an XHTML document. For now, simply copy and paste the **`html`** element start tag on line 8 into your XHTML documents. We discuss the details of the **`html`** element’s **`xmlns`** attribute in Chapter 18, Extensible Markup Language (XML).

Common Programming Error K.1

Not enclosing attribute values in either single or double quotes is a syntax error.

Common Programming Error K.2

Using uppercase letters in an XHTML element or attribute name is a syntax error.

An XHTML document divides the `html` element into two sections—head and body. Lines 9–11 define the Web page's head section with a `head` element. Line 10 specifies a `title` element. This is called a *nested element*, because it is enclosed in the `head` element's start and end tags. The `head` element also is a nested element, because it is enclosed in the `html` element's start and end tags. The `title` element describes the Web page. Titles usually appear in the *title bar* at the top of the browser window and also as the text identifying a page when users add the page to their list of **Favorites** or **Bookmarks**, which enable users to return to their favorite sites. Search engines (i.e., sites that allow users to search the Web) also use the `title` for cataloging purposes.

Good Programming Practice K.3

Indenting nested elements emphasizes a document's structure and promotes readability.

Common Programming Error K.3

XHTML does not permit tags to overlap—a nested element's end tag must appear in the document before the enclosing element's end tag. For example, the nested XHTML tags `<head><title>hello</head></title>` *cause a syntax error, because the enclosing* `head` *element's ending* `</head>` *tag appears before the nested* `title` *element's ending* `</title>` *tag.*

Good Programming Practice K.4

Use a consistent `title` *naming convention for all pages on a site. For example, if a site is named "Bailey's Web Site," then the* `title` *of the main page might be "Bailey's Web Site—Links". This practice can help users better understand the Web site's structure.*

Line 13 opens the document's ***`body`*** element. The body section of an XHTML document specifies the document's content, which may include text and tags.

Some tags, such as the *paragraph tags* (`<p>` and `</p>`) in line 14, mark up text for display in a browser. All text placed between the `<p>` and `</p>` tags form one paragraph. When the browser renders a paragraph, a blank line usually precedes and follows paragraph text.

This document ends with two closing tags (lines 15–16). These tags close the `body` and `html` elements, respectively. The ending `</html>` tag in an XHTML document informs the browser that the XHTML markup is complete.

To view this example in Internet Explorer, perform the following steps:

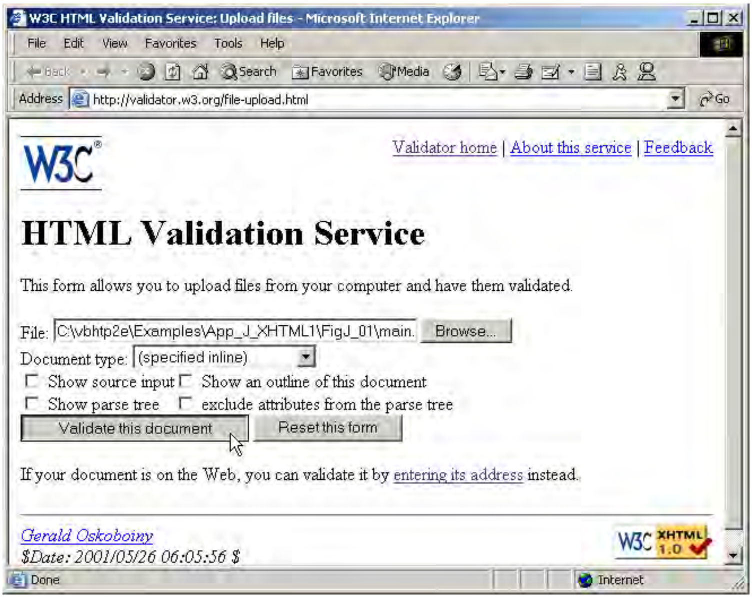
1. Copy the Appendix K examples onto your machine from the CD that accompanies this book (or download the examples from `www.deitel.com`).
2. Launch Internet Explorer, and select **Open...** from the **File** Menu. This displays the **Open** dialog.
3. Click the **Open** dialog's **Browse...** button to display the **Microsoft Internet Explorer** file dialog.
4. Navigate to the directory containing the Appendix K examples and select the file `main.html`; then, click **Open**.
5. Click **OK** to have Internet Explorer render the document. Other examples are opened in a similar manner.

At this point your browser window should appear similar to the sample screen capture shown in Fig. K.1. (Note that we resized the browser window to save space in the book.)

K.4 W3C XHTML Validation Service

Programming Web-based applications can be complex, and XHTML documents must be written correctly to ensure that browsers process them properly. To promote correctly written documents, the World Wide Web Consortium (W3C) provides a *validation service* (`validator.w3.org`) for checking a document's syntax. Documents can be validated either from a URL that specifies the location of the file or by uploading a file to the site `validator.w3.org/file-upload.html`. Uploading a file copies the file from the user's computer to another computer on the Internet. Figure K.2 shows `main.html` (Fig. K.1) being uploaded for validation. Although the W3C's Web page indicates that the service name is **HTML Validation Service**,[1] the validation service is able to validate the syntax of XHTML documents. All the XHTML examples in this book have been validated successfully through `validator.w3.org`.

By clicking **Browse...**, users can select files on their own computers for upload. After selecting a file, clicking the **Validate this document** button uploads and validates the file. Figure 4.3 shows the results of validating `main.html`. This document does not contain any syntax errors. If a document does contain syntax errors, the Validation Service displays error messages describing the errors. In Exercise K.11, we ask readers to create an invalid XHTML document (i.e., one that contains syntax errors) and to check the document's syntax, using the Validation Service. This enables readers to see the types of error messages generated by the validator.


W3C HTML Validation Service: Upload files - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
Media
Address
Go
Validator home | About this service | Feedback
W3C
Browse...
Validate this document
Reset this form
W3C XHTML 1.0
Done
Internet

http://validator.w3.org/file-upload.html

HTML Validation Service

This form allows you to upload files from your computer and have them validated.

File: C:\vbhtp2e\Examples\App_J_XHTML1\FigJ_01\main.

Document type: (specified inline)

☐ Show source input ☐ Show an outline of this document

☐ Show parse tree ☐ exclude attributes from the parse tree

If your document is on the Web, you can validate it by entering its address instead.

Gerald Oskoboiny

$Date: 2001/05/26 06:05:56 $

Fig. K.2 Validating an XHTML document. (Courtesy of World Wide Web Consortium (W3C).)

---

[1] HTML (HyperText Markup Language) is the predecessor of XHTML designed for marking up Web content. HTML is a deprecated technology.

Testing and Debugging Tip K.1

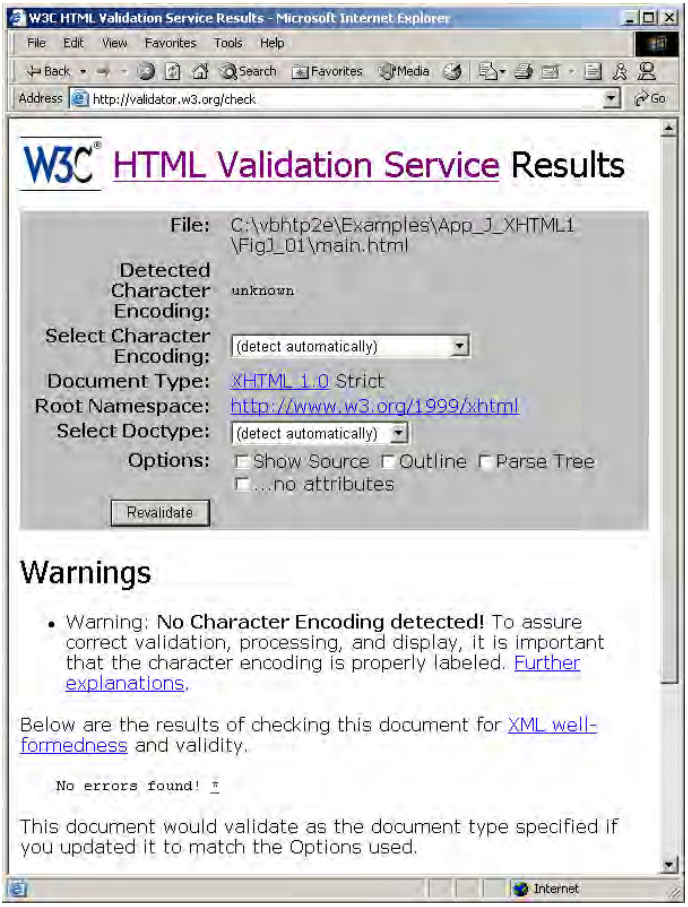
Use a validation service, such as the W3C HTML Validation Service, to confirm that an XHTML document is syntactically correct.

K.5 Headers

Some text in an XHTML document may be more important than some other. For example, the text in this section is considered more important than a footnote. XHTML provides six *headers*, called *header elements*, for specifying the relative importance of information. Figure K.4 demonstrates these elements (**`h1`** through **`h6`**).

Portability Tip K.1

The text size used to display each header element can vary significantly between browsers.


W3C HTML Validation Service Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://validator.w3.org/check
Go
Revalidate
Internet

W3C HTML Validation Service Results

File: C:\vbhtp2e\Examples\App_J_XHTML1\FigJ_01\main.html

Detected Character Encoding: unknown

Select Character Encoding: (detect automatically)

Document Type: XHTML 1.0 Strict

Root Namespace: http://www.w3.org/1999/xhtml

Select Doctype: (detect automatically)

Options: ☐ Show Source ☐ Outline ☐ Parse Tree ☐ ...no attributes

Warnings

- Warning: **No Character Encoding detected!** To assure correct validation, processing, and display, it is important that the character encoding is properly labeled. Further explanations.

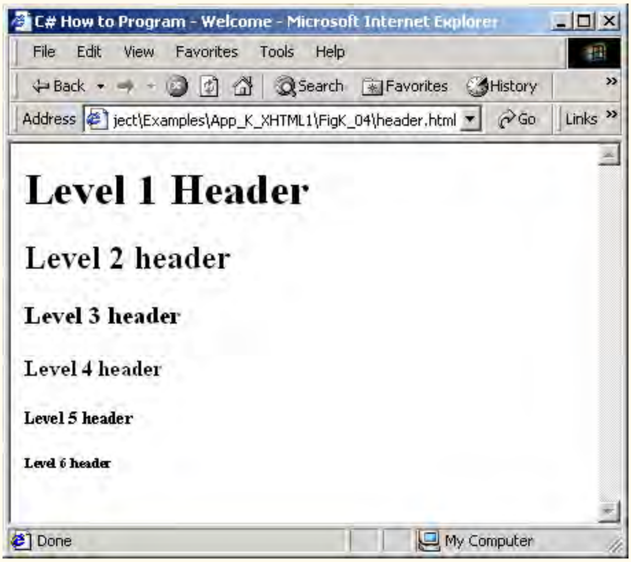
Below are the results of checking this document for XML well-formedness and validity.

```
No errors found! *
```

This document would validate as the document type specified if you updated it to match the Options used.

Fig. K.3 XHTML validation results. (Courtesy of World Wide Web Consortium (W3C).)

```
 1  <?xml version = "1.0"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. K.4: header.html -->
 6  <!-- XHTML headers.         -->
 7
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>C# How to Program - Welcome</title>
11     </head>
12
13     <body>
14
15        <h1>Level 1 Header</h1>
16        <h2>Level 2 header</h2>
17        <h3>Level 3 header</h3>
18        <h4>Level 4 header</h4>
19        <h5>Level 5 header</h5>
20        <h6>Level 6 header</h6>
21
22     </body>
23  </html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address ject\Examples\App_K_XHTML1\FigK_04\header.html Go Links
Done
My Computer

Level 1 Header

Level 2 header

Level 3 header

Level 4 header

Level 5 header

Level 6 header

Fig. K.4 Header elements **h1** through **h6**.

Header element **h1** (line 15) is considered the most significant header and is rendered in a larger font than the other five headers (lines 16–20). Each successive header element (i.e., **h2**, **h3**, etc.) is rendered in a smaller font.

Look-and-Feel Observation K.1

Placing a header at the top of every XHTML page helps viewers understand the purpose of each page.

Look-and-Feel Observation K.2

Use larger headers to emphasize more important sections of a Web page.

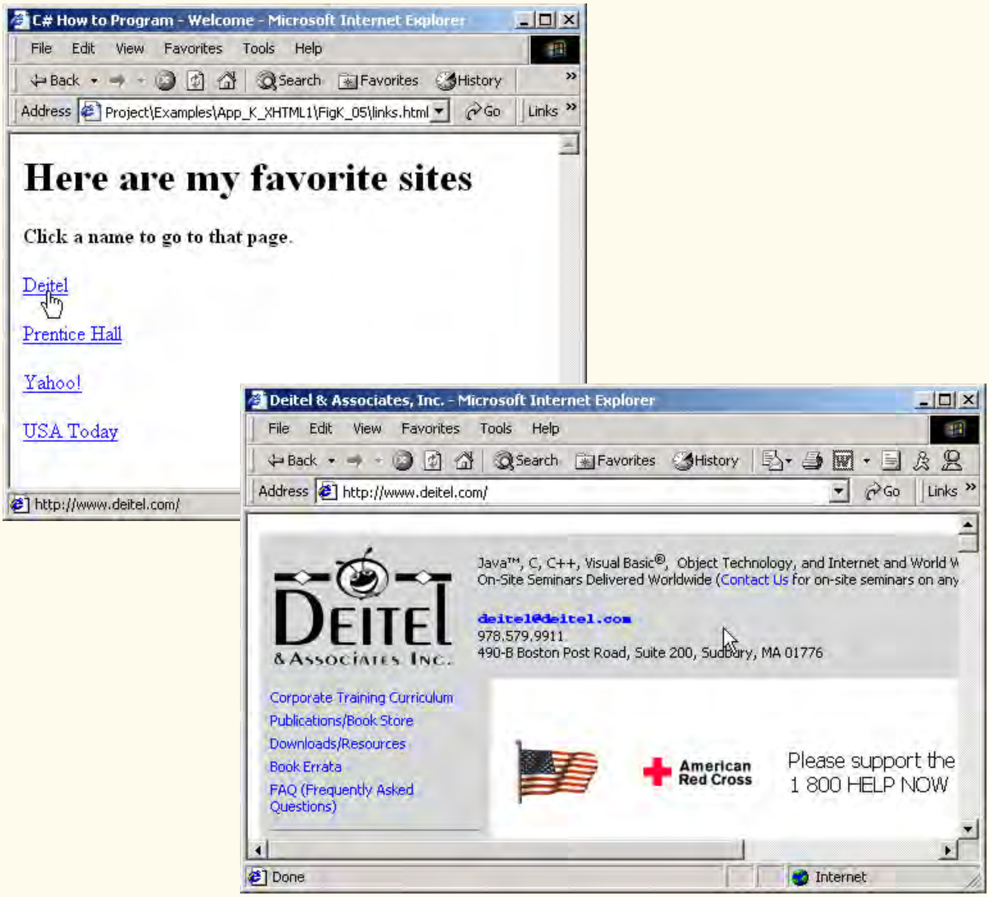
K.6 Linking

One of the most important XHTML features is the *hyperlink*, which references (or *links* to) other resources, such as XHTML documents and images. In XHTML, both text and images can act as hyperlinks. Web browsers typically underline text hyperlinks and color their text blue by default, so that users can distinguish hyperlinks from plain text. In Fig. K.5, we create text hyperlinks to four different Web sites.

Line 17 introduces the **`<strong>`** tag. Browsers typically display text marked up with **`<strong>`** in a bold font.

```
 1  <?xml version = "1.0"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. K.5: links.html        -->
 6  <!-- Introduction to hyperlinks. -->
 7
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>C# How to Program - Welcome</title>
11     </head>
12
13     <body>
14
15        <h1>Here are my favorite sites</h1>
16
17        <p><strong>Click a name to go to that page.</strong></p>
18
19        <!-- create four text hyperlinks -->
20        <p>
21           <a href = "http://www.deitel.com">Deitel</a>
22        </p>
23
24        <p>
25           <a href = "http://www.prenhall.com">Prentice Hall</a>
26        </p>
27
28        <p>
29           <a href = "http://www.yahoo.com">Yahoo!</a>
30        </p>
31
32        <p>
33           <a href = "http://www.usatoday.com">USA Today</a>
34        </p>
35
36     </body>
37  </html>
```

Fig. K.5 Linking to other Web pages. (Part 1 of 2.)


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address Project\Examples\App_K_XHTML1\FigK_05\links.html Go Links
http://www.deitel.com/
Deitel & Associates, Inc. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.deitel.com/ Go Links
DEITEL & ASSOCIATES INC.
American Red Cross
Done
Internet

Here are my favorite sites

Click a name to go to that page.

Deitel

Prentice Hall

Yahoo!

USA Today

Java™, C, C++, Visual Basic®, Object Technology, and Internet and World W
On-Site Seminars Delivered Worldwide (Contact Us for on-site seminars on any

deitel@deitel.com
978.579.9911
490-B Boston Post Road, Suite 200, Sudbury, MA 01776

Corporate Training Curriculum
Publications/Book Store
Downloads/Resources
Book Errata
FAQ (Frequently Asked Questions)

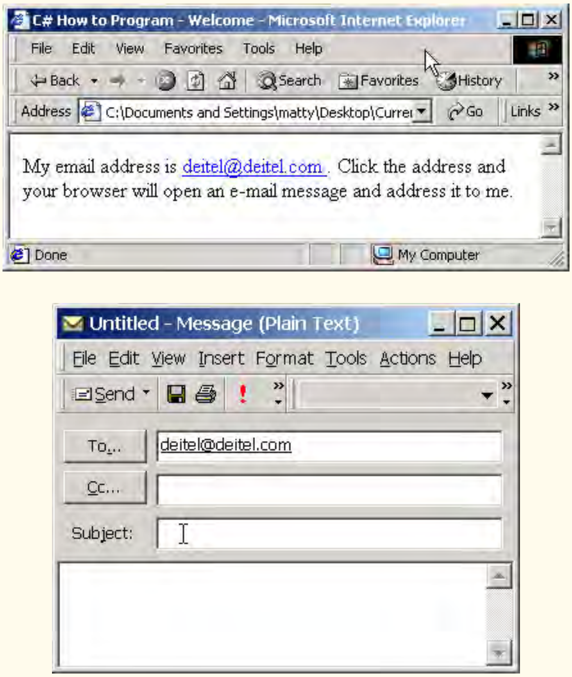
Please support the 1 800 HELP NOW

Fig. K.5 Linking to other Web pages. (Part 2 of 2.)

Links are created using the **`a`** (*anchor*) *element*. Line 21 defines a hyperlink that links the text **`Deitel`** to the URL assigned to attribute ***`href`***, which specifies the location of a linked resource, such as a Web page, a file or an e-mail address. This particular anchor element links to a Web page located at **`http://www.deitel.com`**. When a URL does not indicate a specific document on the Web site, the Web server returns a default Web page. This pages often is called **`index.html`**; however, most Web servers can be configured to to use any file as the default Web page for the site. (Open **`http://www.deitel.com`** in one browser window and **`http://www.deitel.com/index.html`** in a second browser window to confirm that they are identical.) If the Web server cannot locate a requested document, the server returns an error indication to the Web browser and the browser displays an error message to the user.

Anchors can link to e-mail addresses through a ***`mailto:`*** URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to enable the user to write an e-mail message to the linked address. Figure K.6 demonstrates this type of anchor.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. K.6: contact.html    -->
<!-- Adding email hyperlinks. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - Welcome</title>
      </title>
   </head>

   <body>

      <p>My email address is
         <a href = "mailto:deitel@deitel.com">
            deitel@deitel.com
         </a>
         . Click the address and your browser will
         open an e-mail message and address it to me.
      </p>
   </body>
</html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Currei
Go Links
Done
My Computer
Untitled - Message (Plain Text)
File Edit View Insert Format Tools Actions Help
Send
To...
Cc...
Subject:

My email address is deitel@deitel.com. Click the address and your browser will open an e-mail message and address it to me.

To... deitel@deitel.com

Fig. K.6 Linking to an e-mail address.

Lines 17–19 contain an e-mail link. The form of an e-mail anchor is `<a href = "mailto:`*emailaddress*`">…</a>`. In this case, we link to the e-mail address `deitel@deitel.com`.

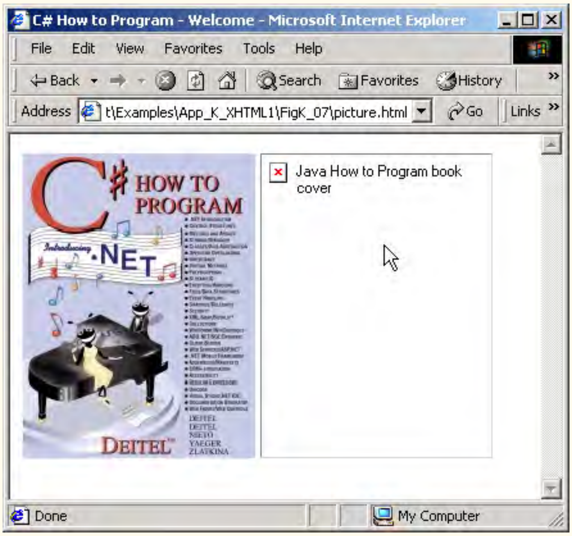
K.7 Images

The examples discussed so far demonstrated how to mark up documents that contain only text. However, most Web pages contain both text and images. In fact, images are an equal and essential part of Web-page design. The two most popular image formats used by Web developers are Graphics Interchange Format (GIF) and Joint Photographic Experts Group (JPEG) images. Users can create images, using specialized pieces of software, such as Adobe PhotoShop Elements and Jasc Paint Shop Pro (`www.jasc.com`). Images may also be acquired from various Web sites, such as `gallery.yahoo.com`. Figure K.7 demonstrates how to incorporate images into Web pages.

Lines 16–17 use an ***`img`*** element to insert an image in the document. The image file's location is specified with the `img` element's ***`src`*** attribute. In this case, the image is located in the same directory as this XHTML document, so only the image's file name is required. Optional attributes ***`width`*** and ***`height`*** specify the image's width and height, respectively. The document author can scale an image by increasing or decreasing the values of the image `width` and `height` attributes. If these attributes are omitted, the browser uses the image's actual width and height. Images are measured in *pixels* ("picture elements"), which represent dots of color on the screen. The image in Fig. K.7 is `181` pixels wide and `238` pixels high.

```
1   <?xml version = "1.0"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4   
5   <!-- Fig. K.7: picture.html    -->
6   <!-- Adding images with XHTML. -->
7   
8   <html xmlns = "http://www.w3.org/1999/xhtml">
9      <head>
10        <title>C# How to Program - Welcome</title>
11     </head>
12  
13     <body>
14  
15        <p>
16           <img src = "csphtp.jpg" height = "238" width = "181"
17              alt = "C# How to Program book cover" />
18  
19           <img src = "jhtp.jpg" height = "238" width = "181"
20              alt = "Java How to Program book cover" />
21        </p>
22  
23     </body>
24  </html>
```

Fig. K.7 Placing images in XHTML files. (Part 1 of 2.)


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address t\Examples\App_K_XHTML1\FigK_07\picture.html Go Links
Done
My Computer

Java How to Program book cover

Fig. K.7 Placing images in XHTML files. (Part 2 of 2.)

Good Programming Practice K.5

Always include the **`width`** *and the* **`height`** *of an image inside the* **`<img>`** *tag. When the browser loads the XHTML file, it will know immediately from these attributes how much screen space to provide for the image and will lay out the page properly, even before it downloads the image.*

Performance Tip K.1

Including the **`width`** *and* **`height`** *attributes in an* **`<img>`** *tag will help the browser load and render pages faster.*

Common Programming Error K.4

Entering new dimensions for an image that change its inherent width-to-height ratio distorts the appearance of the image. For example, if your image is 200 pixels wide and 100 pixels high, you should ensure that any new dimensions have a 2:1 width-to-height ratio.

Every **`img`** element in an XHTML document has an ***`alt`*** attribute. If a browser cannot render an image, the browser displays the **`alt`** attribute's value. A browser might not be able to render an image for several reasons. It might not support images—as is the case with a *text-based browser* (i.e., a browser that can display only text)—or the client may have disabled image viewing to reduce download time. Figure K.7 shows Internet Explorer rendering the **`alt`** attribute's value when a document references a nonexistent image file (**`jhtp.jpg`**).

The **`alt`** attribute is important for creating *accessible* Web pages for users with disabilities, especially those with vision impairments and text-based browsers. Specialized software called *speech synthesizers* often are used by people with disabilities. These software applications "speak" the **`alt`** attribute's value so that the user knows what the browser is displaying. We discuss accessibility issues in detail in Chapter 24, Accessibility.

Some XHTML elements (called *empty elements*) contain only attributes and do not mark up text (i.e., text is not placed between the start and end tags). Empty elements (e.g., `img`) must be terminated, either by using the *forward slash character* (`/`) inside the closing right angle bracket (`>`) of the start tag or by explicitly including the end tag. When using the forward slash character, we add a space before the forward slash to improve readability (as shown at the ends of lines 17 and 20). Rather than using the forward slash character, lines 19–20 could be written with a closing `</img>` tag as follows:

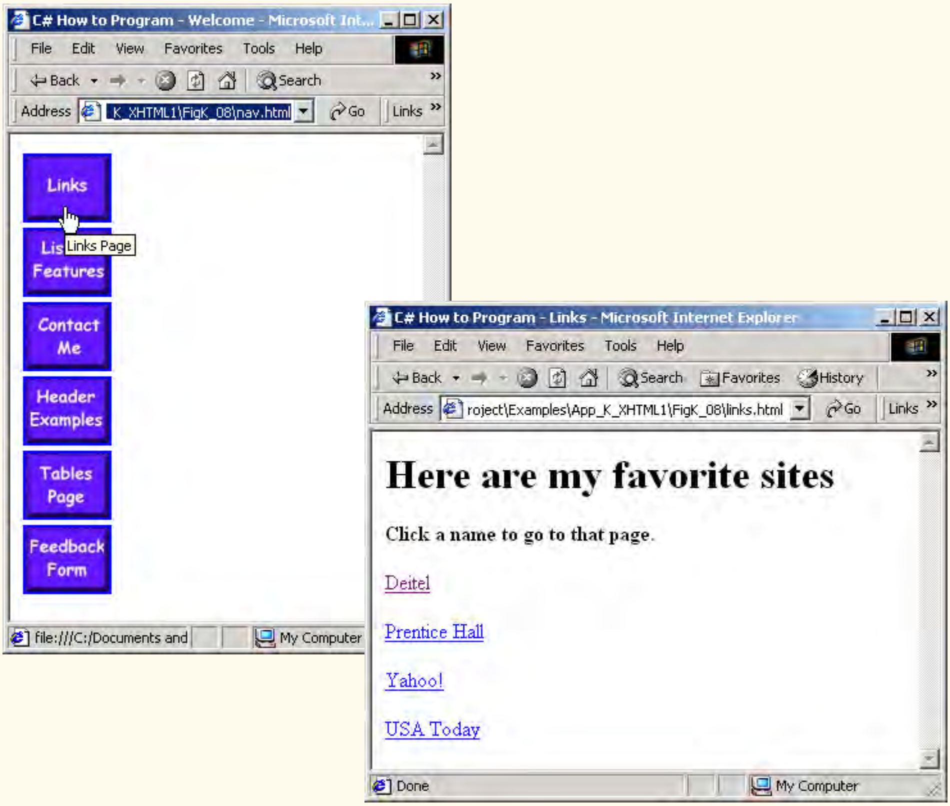
```
<img src = "jhtp.jpg" height = "238" width = "181"
   alt = "Java How to Program book cover"></img>
```

By using images as hyperlinks, Web developers can create graphical Web pages that link to other resources. In Fig. K.8, we create six different image hyperlinks.

```
 1  <?xml version = "1.0"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. K.8: nav.html            -->
 6  <!-- U\sing images as link anchors. -->
 7
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>C# How to Program - Welcome
11        </title>
12     </head>
13
14     <body>
15
16        <p>
17           <a href = "links.html">
18              <img src = "buttons/links.jpg" width = "65"
19                 height = "50" alt = "Links Page" />
20           </a><br />
21
22           <a href = "list.html">
23              <img src = "buttons/list.jpg" width = "65"
24                 height = "50" alt = "List Example Page" />
25           </a><br />
26
27           <a href = "contact.html">
28              <img src = "buttons/contact.jpg" width = "65"
29                 height = "50" alt = "Contact Page" />
30           </a><br />
31
32           <a href = "header.html">
33              <img src = "buttons/header.jpg" width = "65"
34                 height = "50" alt = "Header Page" />
35           </a><br />
36
```

Fig. K.8 Using images as link anchors. (Part 1 of 2.)

```
37             <a href = "table.html">
38                <img src = "buttons/table.jpg" width = "65"
39                   height = "50" alt = "Table Page" />
40             </a><br />
41 
42             <a href = "form.html">
43                <img src = "buttons/form.jpg" width = "65"
44                   height = "50" alt = "Feedback Form" />
45             </a><br />
46          </p>
47 
48       </body>
49    </html>
```


C# How to Program - Welcome - Microsoft Int...
File Edit View Favorites Tools Help
Back Search
Address K_XHTML1\FigK_08\nav.html Go Links
Links
Links Page
List Features
Contact Me
Header Examples
Tables Page
Feedback Form
file:///C:/Documents and
My Computer
C# How to Program - Links - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address roject\Examples\App_K_XHTML1\FigK_08\links.html Go Links
Done
My Computer

Here are my favorite sites

Click a name to go to that page.

Deitel

Prentice Hall

Yahoo!

USA Today

Fig. K.8 Using images as link anchors. (Part 2 of 2.)

Lines 17–20 create an *image hyperlink* by nesting an **`img`** element within an anchor (**`a`**) element. The value of the **`img`** element's **`src`** attribute value specifies that this image (**`links.jpg`**) resides in a directory named **`buttons`**. The **`buttons`** directory and the XHTML document are in the same directory. Images from other Web documents also can be referenced (after obtaining permission from the document's owner) by setting the **`src`** attribute to the name and location of the image.

On line 20, we introduce the ***`br`*** *element*, which most browsers render as a *line break*. Any markup or text following a **`br`** element is rendered on the next line. Like the **`img`** element, **`br`** is an example of an empty element terminated with a forward slash. We add a space before the forward slash to enhance readability.

K.8 Special Characters and More Line Breaks

When marking up text, certain characters or symbols (e.g., **`<`**) may be difficult to embed directly into an XHTML document. Some keyboards may not provide these symbols, or the presence of these symbols may cause syntax errors. For example, the markup

```
<p>if x < 10 then increment x by 1</p>
```

results in a syntax error, because it uses the less-than character (**`<`**), which is reserved for start tags and end tags such as **`<p>`** and **`</p>`**. XHTML provides *special characters* or *entity references* (in the form **`&`***code***`;`**) for representing these characters. We could correct the previous line by writing

```
<p>if x &lt; 10 then increment x by 1</p>
```

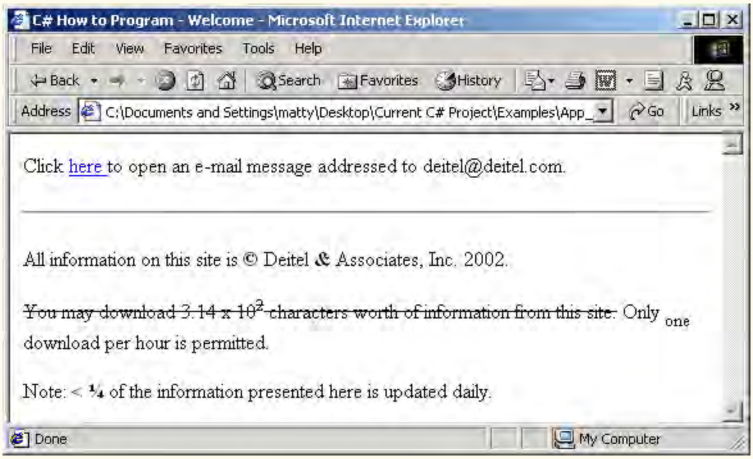
which uses the special character ***`&lt;`*** for the less-than symbol.

Figure K.9 demonstrates how to use special characters in an XHTML document. For a list of special characters, see Appendix M, Special Characters.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. K.9: contact2.html        -->
<!-- Inserting special characters. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - Welcome
      </title>
   </head>

   <body>

      <!-- special characters are    -->
      <!-- entered using form &code; -->
      <p>
         Click
         <a href = "mailto:deitel@deitel.com">here
         </a> to open an e-mail message addressed to
         deitel@deitel.com.
      </p>

      <hr /> <!-- inserts a horizontal rule -->

```

Fig. K.9 Inserting special characters into XHTML. (Part 1 of 2.)

```
27          <p>All information on this site is <strong>&copy;</strong>
28             Deitel <strong>&amp;</strong> Associates, Inc. 2002.</p>
29
30          <!-- to strike through text use <del> tags   -->
31          <!-- to subscript text use <sub> tags        -->
32          <!-- to superscript text use <sup> tags      -->
33          <!-- these tags are nested inside other tags -->
34          <p><del>You may download 3.14 x 10<sup>2</sup>
35             characters worth of information from this site.</del>
36             Only <sub>one</sub> download per hour is permitted.</p>
37
38          <p>Note: <strong>&lt; &frac14;</strong> of the information
39             presented here is updated daily.</p>
40
41       </body>
42    </html>
```


C# How to Program - Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_
Go Links
Done
My Computer

Click here to open an e-mail message addressed to deitel@deitel.com.

All information on this site is © Deitel & Associates, Inc. 2002.

~~You may download 3.14 x 10^2 characters worth of information from this site.~~ Only one download per hour is permitted.

Note: < ¼ of the information presented here is updated daily.

Fig. K.9 Inserting special characters into XHTML. (Part 2 of 2.)

Lines 27–28 contain other special characters, which are expressed as either word abbreviations (e.g., **`amp`** for ampersand and **`copy`** for copyright) or *hexadecimal (hex)* values (e.g., **`&#38;`** is the hexadecimal representation of **`&amp;`**). Hexadecimal numbers are base-16 numbers—digits in a hexadecimal number have values from 0 to 15 (a total of 16 different values). The letters A–F represent the hexadecimal digits corresponding to decimal values 10–15. Thus, in hexadecimal notation we can have numbers like 876 consisting solely of decimal-like digits, numbers like DA19F consisting of digits and letters, and numbers like DCB consisting solely of letters. We discuss hexadecimal numbers in detail in Appendix B, Number Systems.

In lines 34–36, we introduce three new elements. Most browsers render the ***`del`*** element as strike-through text. With this format, users can easily indicate document revisions. To *superscript* text (i.e., raise text on a line with a decreased font size) or *subscript* text (i.e., lower text on a line with a decreased font size), use the ***`sup`*** and ***`sub`*** elements, respectively. We also use special characters **`&lt;`** for a less-than sign and ***`&frac14;`*** for the fraction 1/4 (line 38).

In addition to special characters, this document introduces a *horizontal rule*, indicated by the ***`<hr />`*** tag in line 25. Most browsers render a horizontal rule as a horizontal line. The **`<hr />`** tag also inserts a line break above and below the horizontal line.

K.9 Unordered Lists

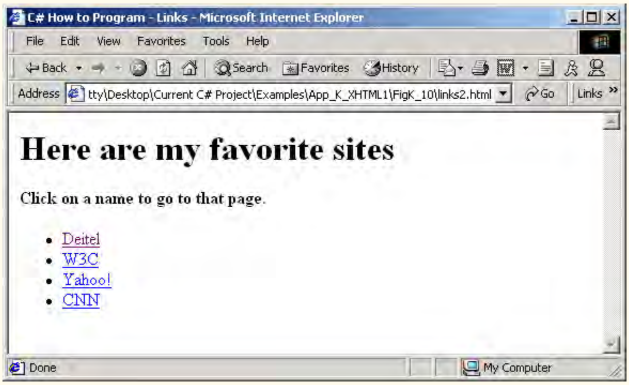
Up to this point, we have presented basic XHTML elements and attributes for linking to resources, creating headers, using special characters and incorporating images. In this section, we discuss how to organize information on a Web page using lists. In Appendix L, Introduction to XHTML: Part 2, we introduce another feature for organizing information, called a table. Figure K.9 displays text in an *unordered list* (i.e., a list that does not order its items by letter or number). The *unordered list element* ***`ul`*** creates a list in which each item begins with a bullet symbol (called a *disc*).

Each entry in an unordered list (element ***`ul`*** in line 20) is an ***`li`*** (*list item*) element (lines 23, 25, 27 and 29). Most Web browsers render these elements with a line break and a bullet symbol indented from the beginning of the new line.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. K.10: links2.html                -->
6  <!-- Unordered list containing hyperlinks. -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>C# How to Program - Links</title>
11    </head>
12
13    <body>
14
15       <h1>Here are my favorite sites</h1>
16
17       <p><strong>Click on a name to go to that page.</strong></p>
18
19       <!-- create an unordered list -->
20       <ul>
21
22          <!-- add four list items -->
23          <li><a href = "http://www.deitel.com">Deitel</a></li>
24
25          <li><a href = "http://www.w3.org">W3C</a></li>
26
27          <li><a href = "http://www.yahoo.com">Yahoo!</a></li>
28
29          <li><a href = "http://www.cnn.com">CNN</a></li>
30
31        </ul>
32
```

Fig. K.9 Unordered lists in XHTML. (Part 1 of 2.)

```
   </body>
</html>
```


C# How to Program - Links - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address tty\Desktop\Current C# Project\Examples\App_K_XHTML1\FigK_10\links2.html
Go
Links
Done
My Computer

Here are my favorite sites

Click on a name to go to that page.

- Deitel
- W3C
- Yahoo!
- CNN

Fig. K.9 Unordered lists in XHTML. (Part 2 of 2.)

K.10 Nested and Ordered Lists

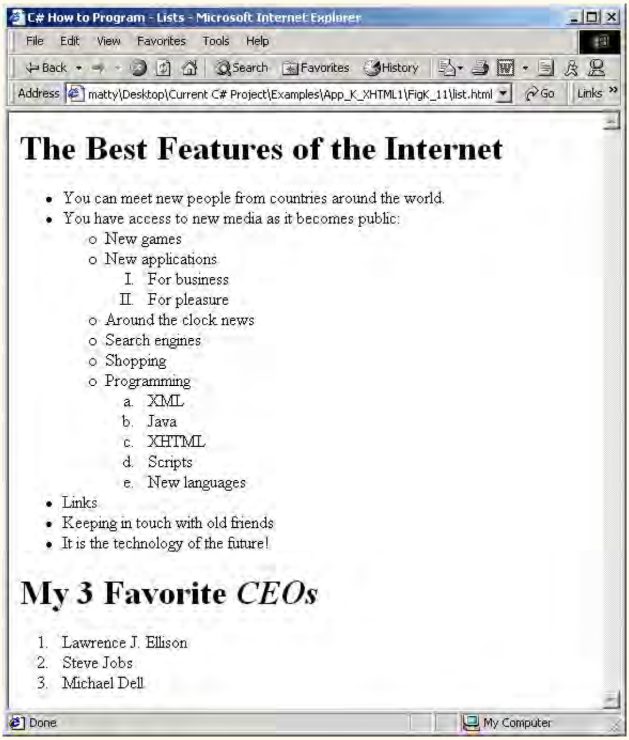
Lists may be nested to represent hierarchical relationships, as in an outline format. Figure K.10 demonstrates nested lists and *ordered lists* (i.e., list that order their items by letter or number).

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. K.11: list.html                 -->
<!-- Advanced Lists: nested and ordered. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - Lists</title>
   </head>

   <body>

      <h1>The Best Features of the Internet</h1>

      <!-- create an unordered list -->
      <ul>
         <li>You can meet new people from countries around
            the world.</li>

         <li>
            You have access to new media as it becomes public:

            <!-- start nested list, use modified bullets -->
            <!-- list ends with closing </ul> tag        -->
```

Fig. K.10 Nested and ordered lists in XHTML. (Part 1 of 3.)

```
27             <ul>
28                <li>New games</li>
29                <li>
30                   New applications
31
32                   <!-- ordered nested list -->
33                   <ol type = "I">
34                      <li>For business</li>
35                      <li>For pleasure</li>
36                   </ol>
37
38                </li>
39
40                <li>Around the clock news</li>
41                <li>Search engines</li>
42                <li>Shopping</li>
43                <li>
44                   Programming
45
46                   <!-- another nested ordered list -->
47                   <ol type = "a">
48                      <li>XML</li>
49                      <li>Java</li>
50                      <li>XHTML</li>
51                      <li>Scripts</li>
52                      <li>New languages</li>
53                   </ol>
54
55                </li>
56
57             </ul> <!-- ends nested list started in line 27 -->
58
59          </li>
60
61          <li>Links</li>
62          <li>Keeping in touch with old friends</li>
63          <li>It is the technology of the future!</li>
64
65       </ul>   <!-- ends unordered list started in line 18 -->
66
67       <h1>My 3 Favorite <em>CEOs</em></h1>
68
69       <!-- ol elements without type attribute have -->
70       <!-- numeric sequence type (i.e., 1, 2, ...) -->
71       <ol>
72          <li>Lawrence J. Ellison</li>
73          <li>Steve Jobs</li>
74          <li>Michael Dell</li>
75       </ol>
76
77    </body>
78 </html>
```

Fig. K.10 Nested and ordered lists in XHTML. (Part 2 of 3.)


C# How to Program - Lists - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address matty\Desktop\Current C# Project\Examples\App_K_XHTML1\FigK_11\list.html
Go
Links
Done
My Computer

The Best Features of the Internet

- You can meet new people from countries around the world.
- You have access to new media as it becomes public:
  - New games
  - New applications
    I. For business
    II. For pleasure
  - Around the clock news
  - Search engines
  - Shopping
  - Programming
    a. XML
    b. Java
    c. XHTML
    d. Scripts
    e. New languages
- Links
- Keeping in touch with old friends
- It is the technology of the future!

My 3 Favorite *CEOs*

1. Lawrence J. Ellison
2. Steve Jobs
3. Michael Dell

Fig. K.10 Nested and ordered lists in XHTML. (Part 3 of 3.)

The first ordered list begins in line 33. Attribute ***type*** specifies the *sequence type* (i.e., the set of numbers or letters used in the ordered list). In this case, setting **type** to **"I"** specifies upper-case roman numerals. Line 47 begins the second ordered list and sets attribute **type** to **"a"**, specifying lowercase letters for the list items. The last ordered list (lines 71–75) does not use attribute **type**. By default, the list's items are enumerated from one to three.

A Web browser indents each nested list to indicate a hierarchal relationship. By default, the items in the outermost unordered list (line 18) are preceded by discs. List items nested inside the unordered list of line 18 are preceded by *circles*. Although not demonstrated in this example, subsequent nested list items are preceded by *squares*. Unordered list items may be explicitly set to discs, circles or squares by setting the **ul** element's **type** attribute to ***"disc"***, ***"circle"*** or ***"square"***, respectively.

Note: XHTML is based on HTML (HyperText Markup Language)—a legacy technology of the World Wide Web Consortium (W3C). In HTML, it was common to specify the document's content, structure and formatting. Formatting might specify where the browser places an element in a Web page or the fonts and colors used to display an element. The so-called *strict* form of XHTML allows only a document's content and structure to appear in a valid XHTML document, and not that document's formatting. Our first several examples used only the strict form of XHTML. In fact, the purpose of lines 2–3 in each of the examples before Fig. K.10 was to indicate to the browser that each document conformed to the strict XHTML definition. This enables the browser to confirm that the document is valid. There are other XHTML document types as well. This particular example uses the XHTML *transitional* document type. This document type exists to enable XHTML document creators to use legacy HTML technologies in an XHTML document. In this example, the **`type`** attribute of the **`ol`** element (lines 33 and 47) is a legacy HTML technology. Changing lines 2–3 as shown in this example, enables us to demonstrate ordered lists with different numbering formats. Normally, such formatting is specified with style sheets. Most examples in this book adhere to strict HTML form.

Testing and Debugging Tip K.2

Most current browsers still attempt to render XHTML documents, even if they are invalid.

K.11 Internet and World Wide Web Resources

`www.w3.org/TR/xhtml1`
The *XHTML 1.0 Recommendation* contains XHTML 1.0 general information, compatibility issues, document type definition information, definitions, terminology and much more.

`www.xhtml.org`
XHTML.org provides XHTML development news and links to other XHTML resources, which include books and articles.

`www.w3schools.com/xhtml/default.asp`
The *XHTML School* provides XHTML quizzes and references. This page also contains links to XHTML syntax, validation and document type definitions.

`validator.w3.org`
This is the W3C XHTML validation service site.

`hotwired.lycos.com/webmonkey/00/50/index2a.html`
This site provides an article about XHTML. Key sections of the article overview XHTML and discuss tags, attributes and anchors.

`wdvl.com/Authoring/Languages/XML/XHTML`
The Web Developers Virtual Library provides an introduction to XHTML. This site also contains articles, examples and links to other technologies.

`www.w3.org/TR/1999/xhtml-modularization-19990406/DTD/doc`
The XHTML 1.0 DTD documentation site provides links to DTD documentation for the strict, transitional and frameset document type definitions.

SUMMARY

- XHTML (Extensible Hypertext Markup Language) is a markup language for creating Web pages.
- A key issue when using XHTML is the separation of the presentation of a document (i.e., the document's appearance when rendered by a browser) from the structure of the information in the document.
- In XHTML, text is marked up with elements, delimited by tags that are names contained in pairs of angle brackets. Some elements may contain additional markup called attributes, which provide additional information about the element.
- A machine that runs specialized piece of software called a Web server stores XHTML documents.
- XHTML documents that are syntactically correct are guaranteed to render properly. XHTML documents that contain syntax errors may not display properly.

- Validation services (e.g., **`validator.w3.org`**) ensure that an XHTML document is syntactically correct.
- Every XHTML document contains a start **`<html>`** tag and an end **`</html>`** tag.
- Comments in XHTML always begin with **`<!--`** and end with **`-->`**. The browser ignores all text inside a comment.
- Every XHTML document has a **`head`** element, which generally contains information, such as a title, and a **`body`** element, which contains the page content. Information in the **`head`** element generally is not rendered in the display window, but it may be made available to the user through other means.
- The **`title`** element names a Web page. The title usually appears in the colored bar (called the title bar) at the top of the browser window and also appears as the text identifying a page when users add your page to their list of **Favorites** or **Bookmarks**.
- The body of an XHTML document is the area in which the document's content is placed. The content may include text and tags.
- All text placed between the **`<p>`** and **`</p>`** tags forms one paragraph.
- XHTML provides six headers (**`h1`** through **`h6`**) for specifying the relative importance of information. Header element **`h1`** is considered the most significant header and is rendered in a larger font than the other five headers. Each successive header element (i.e., **`h2`**, **`h3`**, etc.) is rendered in a smaller font.
- Web browsers typically underline text hyperlinks and color them blue by default.
- The **`<strong>`** tag renders text in a bold font.
- Users can insert links with the **`a`** (anchor) element. The most important attribute for the **`a`** element is **`href`**, which specifies the resource (e.g., page, file or e-mail address) being linked.
- Anchors can link to an e-mail address, using a **`mailto`** URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to initiate an e-mail message to the linked address.
- The **`img`** element's **`src`** attribute specifies an image's location. Optional attributes **`width`** and **`height`** specify the image width and height, respectively. Images are measured in pixels ("picture elements"), which represent dots of color on the screen. Every **`img`** element in a valid XHTML document must have an **`alt`** attribute, which contains text that is displayed if the client cannot render the image.
- The **`alt`** attribute makes Web pages more accessible to users with disabilities, especially those with vision impairments.
- Some XHTML elements are empty elements, contain only attributes and do not mark up text. Empty elements (e.g., **`img`**) must be terminated, either by using the forward slash character (**`/`**) or by explicitly writing an end tag.
- The **`br`** element causes most browsers to render a line break. Any markup or text following a **`br`** element is rendered on the next line.
- XHTML provides special characters or entity references (in the form **`&`***code***`;`**) for representing characters that cannot be marked up.
- Most browsers render a horizontal rule, indicated by the **`<hr />`** tag, as a horizontal line. The **`hr`** element also inserts a line break above and below the horizontal line.
- The unordered list element **`ul`** creates a list in which each item in the list begins with a bullet symbol (called a disc). Each entry in an unordered list is an **`li`** (list item) element. Most Web browsers render these elements with a line break and a bullet symbol at the beginning of the line.
- Lists may be nested to represent hierarchical data relationships.
- Attribute **`type`** specifies the sequence type (i.e., the set of numbers or letters used in the ordered list).

TERMINOLOGY

`<!--…-->` (XHTML comment)
`a` element (`<a>…</a>`)
`alt` attribute
`&amp;` (& special character)
anchor
angle brackets (`< >`)
attribute
`body` element
`br` (line break) element
comments in XHTML
`&copy;` (© special character)
disc
element
e-mail anchor
empty tag
Extensible Hypertext Markup Language (XHTML)
`head` element
header
header elements (`h1` through `h6`)
`height` attribute
hexadecimal code
`<hr />` tag (horizontal rule)
`href` attribute
`.htm` (XHTML file-name extension)
`<html>` tag
`.html` (XHTML file-name extension)
hyperlink
image hyperlink
`img` element
level of nesting
`<li>` (list item) tag
linked document
`mailto:` URL
markup language
nested list
`ol` (ordered list) element
`p` (paragraph) element
special character
`src` attribute (`img`)
`<strong>` tag
`sub` element
subscript
superscript
syntax
tag
text editor
`title` element
`type` attribute
unordered list element (`ul`)
valid document
Web page
`width` attribute
World Wide Web (WWW)
XHTML (Extensible Hypertext Markup Language)
XHTML comment
XHTML markup
XHTML tag
XML declaration
`xmlns` attribute

SELF-REVIEW EXERCISES

K.1 State whether the following statements are *true* or *false*. If *false*, explain why.

a) Attribute `type`, when used with an `ol` element, specifies a sequence type.
b) An ordered list cannot be nested inside an unordered list.
c) XHTML is an acronym for XML HTML.
d) Element `br` represents a line break.
e) Hyperlinks are marked up with `<link>` tags.

K.2 Fill in the blanks in each of the following statements:

a) The __________ element inserts a horizontal rule.
b) A superscript is marked up with element __________ and a subscript is marked up with element __________.
c) The least important header element is __________ and the most important header element is __________.
d) Element __________ marks up an unordered list.
e) Element __________ marks up a paragraph.

ANSWERS TO SELF-REVIEW EXERCISES

K.1 a) True. b) False. An ordered list can be nested inside an unordered list. c) False. XHTML is an acronym for Extensible HyperText Markup Language. d) True. e) False. A hyperlink is marked up with `<a>` tags.

K.2 a) `hr`. b) `sup`, `sub`. c) `h6`, `h1`. d) `ul`. e) `p`.

EXERCISES

K.3 Use XHTML to create a document that contains instructions to mark up the following text:

> Internet and World Wide Web How to Program: Second Edition
> Welcome to the world of Internet programming. We have provided topical coverage for many Internet-related topics.

Use `h1` for the title (the first line of text), `p` for text (the second and third lines of text) and `sub` for each word that begins with a capital letter. Insert a horizontal rule between the `h1` element and the `p` element. Open your new document in a Web browser to view the marked-up document.

K.4 Why is the following markup invalid?

```
<p>Here is some text...
<hr />
<p>And some more text...</p>
```

K.5 Why is the following markup invalid?

```
<p>Here is some text...<br>
And some more text...</p>
```

K.6 An image named `deitel.gif` is 200 pixels wide and 150 pixels high. Use the `width` and `height` attributes of the `<img>` tag to (a) increase the size of the image by 100%; (b) increase the size of the image by 50%; and (c) change the width-to-height ratio to 2:1, keeping the `width` attained in part (a). Write separate XHTML statements for parts (a), (b) and (c).

K.7 Create a link to each of the following: (a) `index.html`, located in the `files` directory; (b) `index.html`, located in the `text` subdirectory of the `files` directory; (c) `index.html`, located in the `other` directory in your *parent directory* [*Hint*: `..` signifies parent directory.]; (d) The United States President's e-mail address (`president@whitehouse.gov`); and (e) An `FTP` link to the file named `README` in the `pub` directory of `ftp.cdrom.com` [*Hint*: Use `ftp://`.].

K.8 Create an XHTML document that marks up your resume.

K.9 Create an XHTML document containing three ordered lists: ice cream, soft serve and frozen yogurt. Each ordered list should contain a nested, unordered list of your favorite flavors. Provide a minimum of three flavors in each unordered list.

K.10 Create an XHTML document that uses an image as an e-mail link. Use attribute `alt` to provide a description of the image and link.

K.11 Create an XHTML document that contains an ordered list of your favorite Web sites. Your page should contain the header "My Favorite Web Sites."

K.12 Create an XHTML document that contains links to all the examples presented in this appendix. [*Hint*: Place all the appendix examples in one directory].

K.13 Modify the XHTML document (`picture.html`) in Fig. K.7 by removing all end tags. Validate this document, using the W3C validation service. What happens? Next remove the `alt` attributes from the `<img>` tags and revalidate your document. What happens?

K.14 Identify each of the following as either an element or an attribute:

a) **`html`**
b) **`width`**
c) **`href`**
d) **`br`**
e) **`h3`**
f) **`a`**
g) **`src`**

K.15 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a) A valid XHTML document can contain uppercase letters in element names.
b) Tags need not be closed in a valid XHTML document.
c) XHTML documents can have the file extension **`.htm`**.
d) Valid XHTML documents can contain tags that overlap.
e) **`&less;`** is the special character for the less-than (**`<`**) character.
f) In a valid XHTML document, **`<li>`** can be nested inside either **`<ol>`** or **`<ul>`** tags.

K.16 Fill in the blanks for each of the following statements:

a) XHTML comments begin with **`<!--`** and end with __________.
b) In XHTML, attribute values must be enclosed in __________.
c) __________ is the special character for an ampersand.
d) Element __________ can be used to bold text.

L

Introduction to XHTML: Part 2

Objectives

- To be able to create tables with rows and columns of data.
- To be able to control table formatting.
- To be able to create and use forms.
- To be able to create and use image maps to aid in Web-page navigation.
- To be able to make Web pages accessible to search engines through **`<meta>`** tags.
- To be able to use the **`frameset`** element to display multiple Web pages in a single browser window.

Yea, from the table of my memory
I'll wipe away all trivial fond records.
William Shakespeare

Outline

L.1 Introduction
L.2 Basic XHTML Tables
L.3 Intermediate XHTML Tables and Formatting
L.4 Basic XHTML Forms
L.5 More Complex XHTML Forms
L.6 Internal Linking
L.7 Creating and Using Image Maps
K.8 `meta` Elements
K.9 `frameset` Element
K.10 Nested `frameset`s
L.11 Internet and World Wide Web Resources
Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

L.1 Introduction

In the previous appendix, we introduced XHTML. We built several complete Web pages featuring text, hyperlinks, images, horizontal rules and line breaks. In this appendix, we discuss more substantial XHTML features, including presentation of information in *tables* and *incorporating forms* for collecting information from a Web-page visitor. We also introduce *internal linking* and *image maps* for enhancing Web-page navigation and *frames* for displaying multiple documents in the browser. By the end of this appendix, you will be familiar with the most commonly used XHTML features and will be able to create more complex Web documents. In this appendix, we do not present any C# programming.

L.2 Basic XHTML Tables

This section presents the XHTML *table*—a frequently used feature that organizes data into rows and columns. Our first example (Fig. L.1) uses a table with six rows and two columns to display price information for fruit.

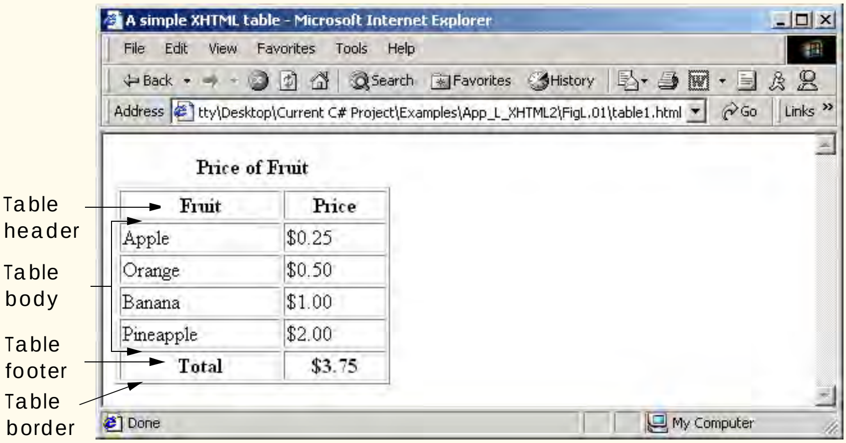
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. L.1: table1.html   -->
<!-- Creating a basic table. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>A simple XHTML table</title>
   </head>

```

Fig. L.1 XHTML table. (Part 1 of 3.)

```
   <body>

      <!-- the <table> tag begins table -->
      <table border = "1" width = "40%"
         summary = "This table provides information about
            the price of fruit">

         <!-- <caption> tag summarizes table's   -->
         <!-- contents to help visually impaired -->
         <caption><strong>Price of Fruit</strong></caption>

         <!-- <thead> is first section of table -->
         <!-- it formats table header area      -->
         <thead>
            <tr> <!-- <tr> inserts one table row -->
               <th>Fruit</th> <!-- insert heading cell -->
               <th>Price</th>
            </tr>
         </thead>

         <!-- all table content is enclosed within <tbody> -->
         <tbody>
            <tr>
               <td>Apple</td> <!-- insert data cell -->
               <td>$0.25</td>
            </tr>

            <tr>
               <td>Orange</td>
               <td>$0.50</td>
            </tr>

            <tr>
               <td>Banana</td>
               <td>$1.00</td>
            </tr>

            <tr>
               <td>Pineapple</td>
               <td>$2.00</td>
            </tr>
         </tbody>

         <!-- <tfoot> is last section of table -->
         <!-- it formats table footer          -->
         <tfoot>
            <tr>
               <th>Total</th>
               <th>$3.75</th>
            </tr>
         </tfoot>

      </table>
```

Fig. L.1 XHTML table. (Part 2 of 3.)

```
66
67       </body>
68    </html>
```


A simple XHTML table - Microsoft Internet Explorer
File
Edit
View
Favorites
Tools
Help
Back
Search
Favorites
History
Address
tty\Desktop\Current C# Project\Examples\App_L_XHTML2\FigL.01\table1.html
Go
Links
Done
My Computer
Table header
Table body
Table footer
Table border

Price of Fruit

Fruit	Price
Apple	$0.25
Orange	$0.50
Banana	$1.00
Pineapple	$2.00
Total	**$3.75**

Fig. L.1 XHTML table. (Part 3 of 3.)

Tables are defined with the **`table`** element. Lines 16–18 specify the start tag for a table element that has several attributes. The ***`border`*** attribute specifies the table's border width in pixels. To create a table without a border, set **`border`** to **`"0"`**. This example assigns attribute **`width "40%"`**, to set the table's width to 40 percent of the browser's width. A developer can also set attribute **`width`** to a specified number of pixels.

Testing and Debugging Tip L.1

Try resizing the browser window to see how the width of the window affects the width of the table.

As its name implies, attribute ***`summary`*** (line 17) describes the table's contents. Speech devices use this attribute to make the table more accessible to users with visual impairments. The ***`caption`*** element (line 22) describes the table's content and helps text-based browsers interpret the table data. Text inside the **`<caption>`** tag is rendered above the table by most browsers. Attribute **`summary`** and element **`caption`** are two of many XHTML features that make Web pages more accessible to users with disabilities. We discuss accessibility programming in detail in Chapter 24, Accessibility.

A table has three distinct sections—*head*, *body* and *foot*. The head section (or *header cell*) is defined with a ***`thead`*** element (lines 26–31), which contains header information, such as column names. Each ***`tr`*** element (lines 27–30) defines an individual *table row*. The columns in the head section are defined with **`th`** elements. Most browsers center text formatted by **`th`** (table header column) elements and display it in bold. Table header elements are nested inside table row elements.

The body section, or *table body*, contains the table's primary data. The table body (lines 34–54) is defined in a **`tbody`** element. *Data cells* contain individual pieces of data and are defined with **`td`** (*table data*) elements.

The foot section (lines 58–63) is defined with a `tfoot` (table foot) element and represents a footer. Text commonly placed in the footer includes calculation results and footnotes. Like other sections, the foot section can contain table rows and each row can contain columns.

L.3 Intermediate XHTML Tables and Formatting

In the previous section, we explored the structure of a basic table. In Fig. L.2, we enhance our discussion of tables by introducing elements and attributes that allow the document author to build more complex tables.

The table begins on line 17. Element `colgroup` (lines 22–27) groups and formats columns. The `col` element (line 26) specifies two attributes in this example. The `align` attribute determines the alignment of text in the column. The `span` attribute determines how many columns the `col` element formats. In this case, we set `align`'s value to `"right"` and `span`'s value to `"1"` to right-align text in the first column (the column containing the picture of the camel in the sample screen capture).

Table cells are sized to fit the data they contain. Document authors can create larger data cells by using attributes `rowspan` and `colspan`. The values assigned to these attributes specify the number of rows or columns occupied by a cell. The `th` element at lines 36–39 uses the attribute `rowspan = "2"` to allow the cell containing the picture of the camel to use two vertically adjacent cells (thus the cell *spans* two rows). The `th` element at lines 42–45 uses the attribute `colspan = "4"` to widen the header cell (containing `Camelid comparison` and `Approximate as of 9/2002`) to span four cells.

```
1   <?xml version = "1.0"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. L.2: table2.html      -->
6   <!-- Intermediate table design. -->
7
8   <html xmlns = "http://www.w3.org/1999/xhtml">
9      <head>
10        <title>C# How to Program - Tables</title>
11     </head>
12
13     <body>
14
15        <h1>Table Example Page</h1>
16
17        <table border = "1">
18           <caption>Here is a more complex sample table.</caption>
19
20           <!-- <colgroup> and <col> tags are -->
21           <!-- used to format entire columns -->
22           <colgroup>
23
24              <!-- span attribute determines how   -->
25              <!-- many columns <col> tag affects -->
```

Fig. L.2 Complex XHTML table. (Part 1 of 3.)

```
            <col align = "right" span = "1" />
         </colgroup>

         <thead>

            <!-- rowspans and colspans merge specified    -->
            <!-- number of cells vertically or horizontally   -->
            <tr>

               <!-- merge two rows -->
               <th rowspan = "2">
                  <img src = "camel.gif" width = "205"
                     height = "167" alt = "Picture of a camel" />
               </th>

               <!-- merge four columns -->
               <th colspan = "4" valign = "top">
                  <h1>Camelid comparison</h1><br />
                  <p>Approximate as of 9/2002</p>
               </th>
            </tr>

            <tr valign = "bottom">
               <th># of Humps</th>
               <th>Indigenous region</th>
               <th>Spits?</th>
               <th>Produces Wool?</th>
            </tr>

         </thead>

         <tbody>

            <tr>
               <th>Camels (bactrian)</th>
               <td>2</td>
               <td>Africa/Asia</td>
               <td rowspan = "2">Llama</td>
               <td rowspan = "2">Llama</td>
            </tr>

            <tr>
               <th>Llamas</th>
               <td>1</td>
               <td>Andes Mountains</td>
            </tr>

         </tbody>

      </table>

   </body>
</html>
```

Fig. L.2 Complex XHTML table. (Part 2 of 3.)

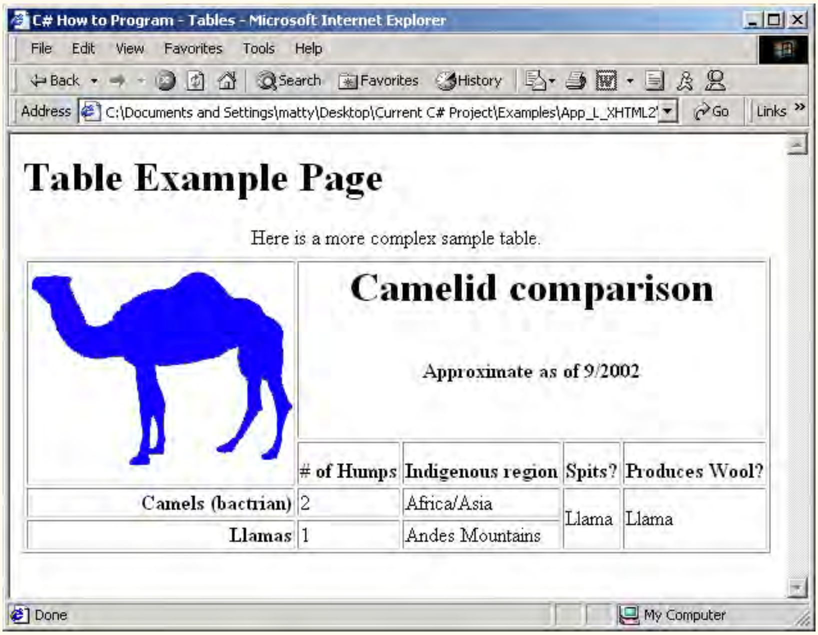

C# How to Program - Tables - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_L_XHTML2'
Go
Links
Done
My Computer

Table Example Page

Here is a more complex sample table.

	Camelid comparison Approximate as of 9/2002			
	# of Humps	Indigenous region	Spits?	Produces Wool?
Camels (bactrian)	2	Africa/Asia	Llama	Llama
Llamas	1	Andes Mountains		

Fig. L.2 Complex XHTML table. (Part 3 of 3.)

Common Programming Error L.1

When using **`colspan`** *and* **`rowspan`** *to adjust the size of table data cells, keep in mind that the modified cells will occupy more than one column or row; other rows or columns of the table must compensate for the extra rows or columns spanned by individual cells. If you do not, the formatting of your table will be distorted, and you could inadvertently create more columns and rows than you originally intended.*

Line 42 introduces attribute **`valign`**, which aligns data vertically and may be assigned one of four values—**`"top"`** aligns data with the top of the cell, **`"middle"`** vertically centers data (the default for all data and header cells), **`"bottom"`** aligns data with the bottom of the cell and **`"baseline"`** ignores the fonts used for the row data and sets the bottom of all text in the row on a common *baseline* (i.e., the horizontal line to which each character in a word is aligned).

L.4 Basic XHTML Forms

When browsing Web sites, users often need to provide information such as e-mail addresses, search keywords and zip codes. XHTML provides a mechanism, called a *form*, for collecting such user information.

Data that users enter on a Web page normally is sent to a Web server that provides access to a site's resources (e.g., XHTML documents or images). These resources are located either on the same machine as the Web server or on a machine that the Web server can access through the network. When a browser requests a Web page or file that is located on a server, the server processes the request and returns the requested resource. A request contains the name and path of the desired resource and the method of communication (called a *protocol*). XHTML documents use the HyperText Transfer Protocol (HTTP).

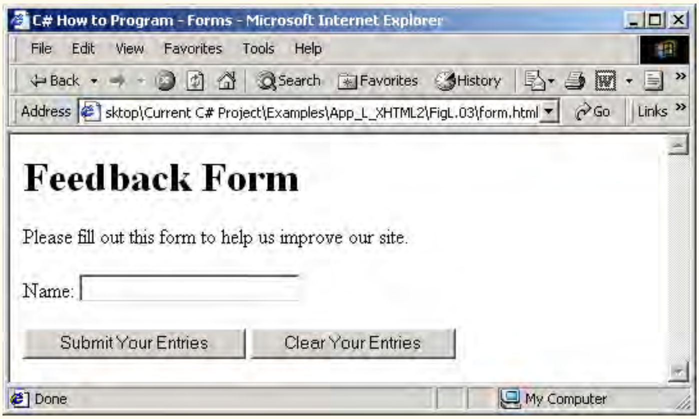
Figure L.3 sends the form data to the Web server, which passes the form data to a *CGI* (*Common Gateway Interface*) script (i.e., a program) written in Perl, C or some other language. The script processes the data received from the Web server and typically returns information to the Web server. The Web server then sends the information in the form of an XHTML document to the Web browser. [*Note*: This example demonstrates client-side functionality. If the form is submitted (by clicking **Submit Your Entries**), an error occurs.]

Forms can contain visual and non-visual components. Visual components include clickable buttons and other graphical user interface components with which users interact. Non-visual components, called *hidden inputs*, store any data that the document author specifies, such as e-mail addresses and XHTML document file names that act as links. The form begins on line 23 with the ***`form`*** element. Attribute ***`method`*** specifies how the form's data is sent to the Web server.

```
1   <?xml version = "1.0"?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5   <!-- Fig. L.3: form.html    -->
6   <!-- Form design example 1. -->
7
8   <html xmlns = "http://www.w3.org/1999/xhtml">
9      <head>
10        <title>C# How to Program - Forms</title>
11     </head>
12
13     <body>
14
15        <h1>Feedback Form</h1>
16
17        <p>Please fill out this form to help
18           us improve our site.</p>
19
20        <!-- <form> tag begins form, gives -->
21        <!-- method of sending information -->
22        <!-- and location of form scripts  -->
23        <form method = "post" action = "/cgi-bin/formmail">
24
25           <p>
26
27              <!-- hidden inputs contain non-visual -->
28              <!-- information                     -->
29              <input type = "hidden" name = "recipient"
30                 value = "deitel@deitel.com" />
31
32              <input type = "hidden" name = "subject"
33                 value = "Feedback Form" />
34
```

Fig. L.3 Simple form with hidden fields and a textbox. (Part 1 of 2.)

```
            <input type = "hidden" name = "redirect"
               value = "main.html" />
         </p>

         <!-- <input type = "text"> inserts text box -->
         <p>
            <label>Name:
               <input name = "name" type = "text" size = "25"
                  maxlength = "30" />
            </label>
         </p>

         <p>

            <!-- input types "submit" and "reset" -->
            <!-- insert buttons for submitting    -->
            <!-- and clearing form's contents     -->
            <input type = "submit" value =
               "Submit Your Entries" />

            <input type = "reset" value =
               "Clear Your Entries" />
         </p>

      </form>

   </body>
</html>
```


C# How to Program - Forms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address sktop\Current C# Project\Examples\App_L_XHTML2\FigL.03\form.html
Go Links
Submit Your Entries
Clear Your Entries
Done
My Computer

Feedback Form

Please fill out this form to help us improve our site.

Name:

Fig. L.3 Simple form with hidden fields and a textbox. (Part 2 of 2.)

Using **`method = "post"`** appends form data to the browser request, which contains the protocol (i.e., HTTP) and the requested resource's URL. Scripts located on the Web server's computer (or on a computer accessible through the network) can access the form data sent as part of the request. For example, a script may take the form information and update an electronic mailing list. The other possible value, **`method = "get"`**, appends the form data directly to the end of the URL. For example, the URL `/cgi-bin/formmail` might have the form information `name = bob` appended to it.

The `action` attribute in the `<form>` tag specifies the URL of a script on the Web server; in this case, it specifies a script that e-mails form data to an address. Most Internet Service Providers (ISPs) have a script like this on their site; ask the Web-site system administrator how to set up an XHTML document to use the script correctly.

Lines 29–36 define three `input` elements that specify data to provide to the script that processes the form (also called the *form handler*). These three `input` element have `type` attribute `"hidden"`, which allows the document author to send form data that is not entered by a user to a script.

The three hidden inputs are an e-mail address to which the data will be sent, the e-mail's subject line and a URL where the browser will be redirected after submitting the form. Two other `input` attributes are `name`, which identifies the `input` element, and `value`, which provides the value that will be sent (or posted) to the Web server.

Good Programming Practice L.1

Place hidden `input` elements at the beginning of a form, immediately after the opening `<form>` tag. This placement allows document authors to locate hidden `input` elements quickly.

We introduce another `type` of `input` in lines 38–39. The `"text"` `input` inserts a *text box* into the form. Users can type data in text boxes. The `label` element (lines 37–40) provides users with information about the `input` element's purpose.

Common Programming Error L.2

Forgetting to include a `label` element for each form element is a design error. Without these labels, users cannot determine the purpose of individual form elements.

The `input` element's `size` attribute specifies the number of characters visible in the text box. Optional attribute `maxlength` limits the number of characters input into the text box. In this case, the user is not permitted to type more than `30` characters into the text box.

There are two types of `input` elements in lines 52–56. The `"submit"` `input` element is a button. When the user presses a `"submit"` button, the browser sends the data in the form to the Web server for processing. The `value` *attribute* sets the text displayed on the button (the default value is **Submit**). The `"reset"` `input` element allows a user to reset all `form` elements to their default values. The `value` attribute of the `"reset"` `input` element sets the text displayed on the button (the default value is **Reset**).

L.5 More Complex XHTML Forms

In the previous section, we introduced basic forms. In this section, we introduce elements and attributes for creating more complex forms. Figure L.4 contains a form that solicits user feedback about a Web site.

The `textarea` element (lines 42–44) inserts a multiline text box, called a *textarea*, into the form. The number of rows is specified with the `rows` *attribute* and the number of columns (i.e., characters) is specified with the `cols` *attribute*. In this example, the `textarea` is four rows high and 36 characters wide. To display default text in the text area, place the text between the `<textarea>` and `</textarea>` tags. Default text can be specified in other `input` types, such as textboxes, by using the `value` attribute.

The ***"password"*** input in lines 52–53 inserts a password box with the specified **`size`**. A password box allows users to enter sensitive information, such as credit card numbers and passwords, by “masking” the information input with asterisks. The actual value input is sent to the Web server, not the asterisks that mask the input.

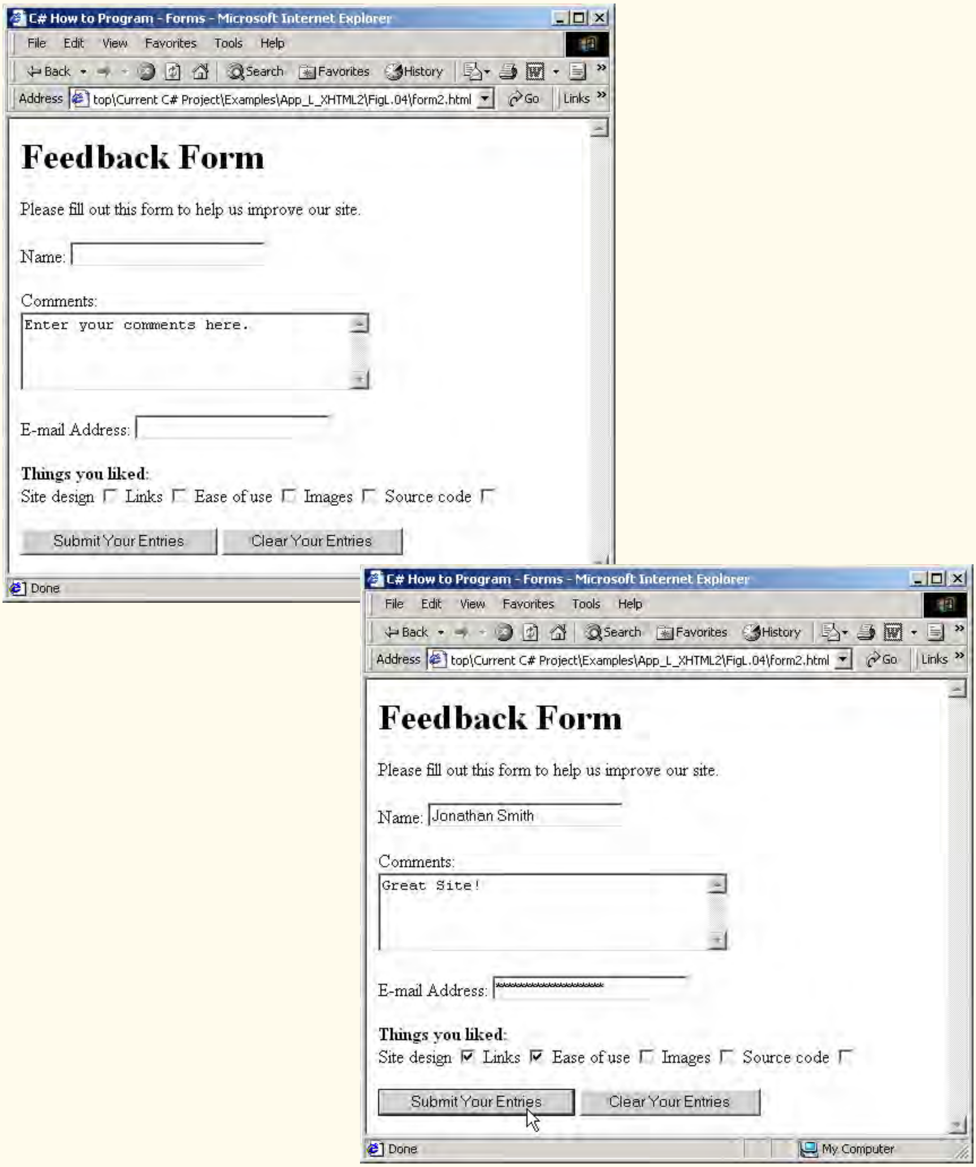
Lines 60–78 introduce the *checkbox* **`form`** element. Checkboxes enable users to select from a set of options. When a user selects a checkbox, a check mark appears in the check box. Otherwise, the checkbox remains empty. Each ***"checkbox"*** **`input`** creates a new checkbox. Checkboxes can be used individually or in groups. Checkboxes that belong to a group are assigned the same **`name`** (in this case, **`"thingsliked"`**).

```
 1  <?xml version = "1.0"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. L.4: form2.html   -->
 6  <!-- Form design example 2. -->
 7
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>C# How to Program - Forms</title>
11     </head>
12
13     <body>
14
15        <h1>Feedback Form</h1>
16
17        <p>Please fill out this form to help
18           us improve our site.</p>
19
20        <form method = "post" action = "/cgi-bin/formmail">
21
22           <p>
23              <input type = "hidden" name = "recipient"
24                 value = "deitel@deitel.com" />
25
26              <input type = "hidden" name = "subject"
27                 value = "Feedback Form" />
28
29              <input type = "hidden" name = "redirect"
30                 value = "main.html" />
31           </p>
32
33           <p>
34              <label>Name:
35                 <input name = "name" type = "text" size = "25" />
36              </label>
37           </p>
38
39           <!-- <textarea> creates multiline textbox -->
40           <p>
41              <label>Comments:<br />
```

Fig. L.4 Form with textareas, password boxes and checkboxes. (Part 1 of 3.)

```
               <textarea name = "comments" rows = "4"
                  cols = "36">Enter your comments here.
               </textarea>
            </label></p>

         <!-- <input type = "password"> inserts -->
         <!-- textboxwhose display is masked     -->
         <!-- with asterisk characters         -->
         <p>
            <label>E-mail Address:
               <input name = "email" type = "password"
                  size = "25" />
            </label>
         </p>

         <p>
            <strong>Things you liked:</strong><br />

            <label>Site design
            <input name = "thingsliked" type = "checkbox"
               value = "Design" /></label>

            <label>Links
            <input name = "thingsliked" type = "checkbox"
               value = "Links" /></label>

            <label>Ease of use
            <input name = "thingsliked" type = "checkbox"
               value = "Ease" /></label>

            <label>Images
            <input name = "thingsliked" type = "checkbox"
               value = "Images" /></label>

            <label>Source code
            <input name = "thingsliked" type = "checkbox"
               value = "Code" /></label>
         </p>

         <p>
            <input type = "submit" value =
               "Submit Your Entries" />

            <input type = "reset" value =
               "Clear Your Entries" />
         </p>

      </form>

   </body>
</html>
```

Fig. L.4 Form with textareas, password boxes and checkboxes. (Part 2 of 3.)


C# How to Program - Forms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address top\Current C# Project\Examples\App_L_XHTML2\FigL.04\form2.html
Go Links
Submit Your Entries
Clear Your Entries
Done
My Computer

Feedback Form

Please fill out this form to help us improve our site.

Name:

Comments:

Enter your comments here.

E-mail Address:

Things you liked:
Site design ☐ Links ☐ Ease of use ☐ Images ☐ Source code ☐

Feedback Form

Please fill out this form to help us improve our site.

Name: Jonathan Smith

Comments:

Great Site!

E-mail Address: ********************

Things you liked:
Site design ☑ Links ☑ Ease of use ☐ Images ☐ Source code ☐

Fig. L.4 Form with textareas, password boxes and checkboxes. (Part 3 of 3.)

Common Programming Error L.3

When your **`form`** *has several checkboxes with the same* **`name`***, you must make sure that they have different* **`value`***s, or the scripts running on the Web server will not be able to distinguish between them.*

We continue our discussion of forms by presenting a third example that introduces several more form elements from which users can make selections (Fig. L.5). In this example, we introduce two new **`input`** types. The first type is the *radio button* (lines 90–113), specified with type **`"radio"`**. Radio buttons are similar to checkboxes, except that only one radio button in a group of radio buttons may be selected at any time. All radio buttons in a group have the same **`name`** attribute; they are distinguished by their different **`value`** attributes. The attribute–value pair ***`checked = "checked"`*** (line 92) indicates which radio button, if any, is selected initially. The **`checked`** attribute also applies to checkboxes.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. L.5: form3.html   -->
6  <!-- Form design example 3. -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>C# How to Program - Forms</title>
11    </head>
12
13    <body>
14
15       <h1>Feedback Form</h1>
16
17       <p>Please fill out this form to help
18          us improve our site.</p>
19
20       <form method = "post" action = "/cgi-bin/formmail">
21
22          <p>
23             <input type = "hidden" name = "recipient"
24                value = "deitel@deitel.com" />
25
26             <input type = "hidden" name = "subject"
27                value = "Feedback Form" />
28
29             <input type = "hidden" name = "redirect"
30                value = "main.html" />
31          </p>
32
33          <p>
34             <label>Name:
35                <input name = "name" type = "text" size = "25" />
36             </label>
37          </p>
38
39          <p>
40             <label>Comments:<br />
41                <textarea name = "comments" rows = "4"
42                   cols = "36"></textarea>
43             </label>
44          </p>
45
```

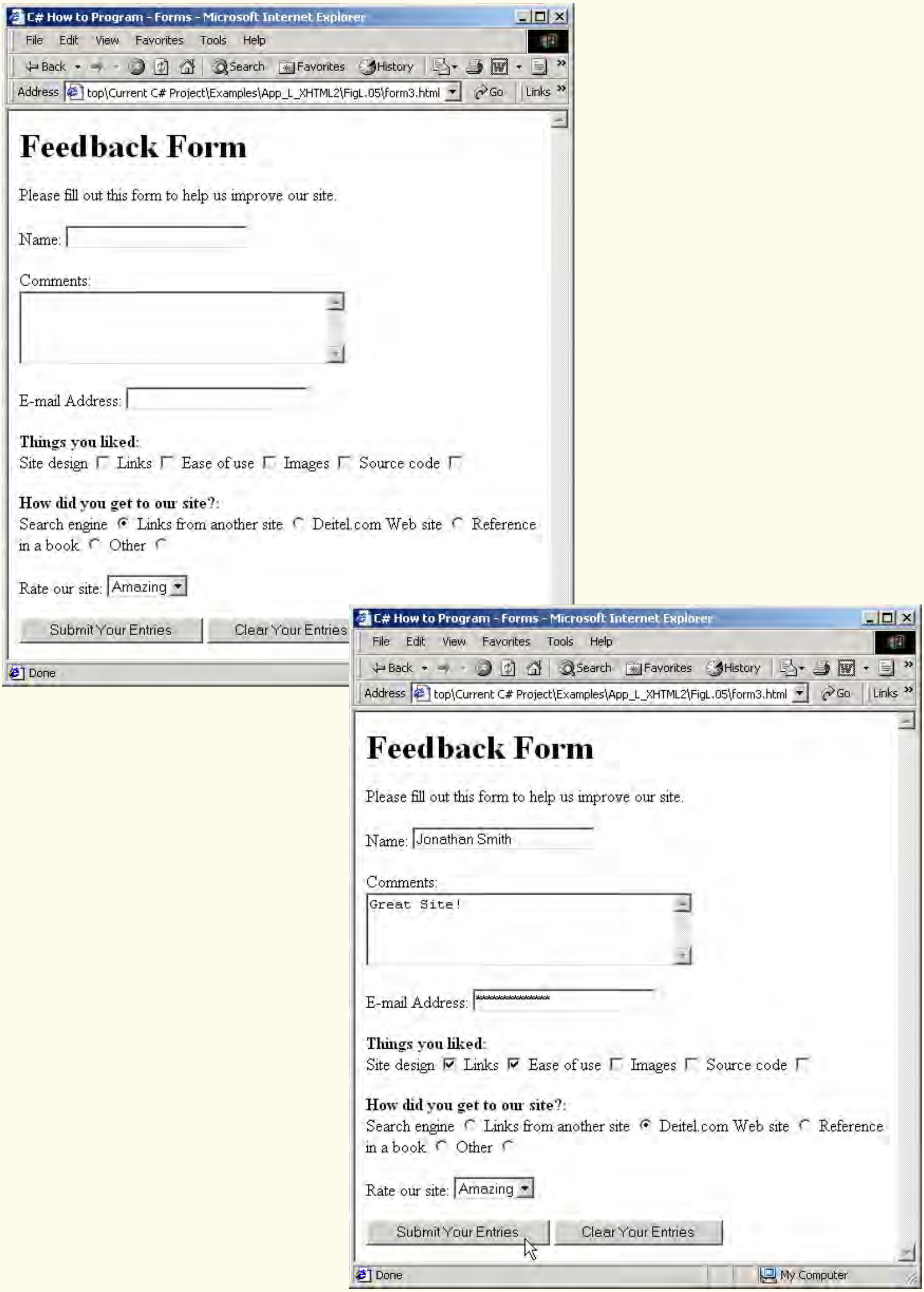
Fig. L.5 Form including radio buttons and drop-down lists. (Part 1 of 4.)

```
         <p>
            <label>E-mail Address:
               <input name = "email" type = "password"
                  size = "25" />
            </label>
         </p>

         <p>
            <strong>Things you liked:</strong><br />

            <label>Site design
               <input name = "thingsliked" type = "checkbox"
                  value = "Design" />
            </label>

            <label>Links
               <input name = "thingsliked" type = "checkbox"
                  value = "Links" />
            </label>

            <label>Ease of use
               <input name = "thingsliked" type = "checkbox"
                  value = "Ease" />
            </label>

            <label>Images
               <input name = "thingsliked" type = "checkbox"
                  value = "Images" />
            </label>

            <label>Source code
               <input name = "thingsliked" type = "checkbox"
                  value = "Code" />
            </label>

         </p>

         <!-- <input type = "radio" /> creates one radio   -->
         <!-- button. The difference between radio buttons -->
         <!-- and checkboxes is that only one radio button -->
         <!-- in a group can be selected.                 -->
         <p>
            <strong>How did you get to our site?:</strong><br />

            <label>Search engine
               <input name = "howtosite" type = "radio"
                  value = "search engine" checked = "checked" />
            </label>

            <label>Links from another site
               <input name = "howtosite" type = "radio"
                  value = "link" />
            </label>
```

Fig. L.5 Form including radio buttons and drop-down lists. (Part 2 of 4.)

```

               <label>Deitel.com Web site
                  <input name = "howtosite" type = "radio"
                     value = "deitel.com" />
               </label>

               <label>Reference in a book
                  <input name = "howtosite" type = "radio"
                     value = "book" />
               </label>

               <label>Other
                  <input name = "howtosite" type = "radio"
                     value = "other" />
               </label>

            </p>

            <p>
               <label>Rate our site:

                   <!-- <select> tag presents a drop-down -->
                   <!-- list with choices indicated by    -->
                   <!-- <option> tags                     -->
                  <select name = "rating">
                     <option selected = "selected">Amazing</option>
                     <option>10</option>
                     <option>9</option>
                     <option>8</option>
                     <option>7</option>
                     <option>6</option>
                     <option>5</option>
                     <option>4</option>
                     <option>3</option>
                     <option>2</option>
                     <option>1</option>
                     <option>Awful</option>
                  </select>

               </label>
            </p>

            <p>
               <input type = "submit" value =
                  "Submit Your Entries" />

               <input type = "reset" value = "Clear Your Entries" />
            </p>

         </form>

      </body>
   </html>
```

Fig. L.5 Form including radio buttons and drop-down lists. (Part 3 of 4.)


C# How to Program - Forms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address top\Current C# Project\Examples\App_L_XHTML2\FigL.05\form3.html
Go Links
Done
My Computer

Feedback Form

Please fill out this form to help us improve our site.

Name:

Comments:

E-mail Address:

Things you liked:
Site design ☐ Links ☐ Ease of use ☐ Images ☐ Source code ☐

How did you get to our site?:
Search engine ● Links from another site ○ Deitel.com Web site ○ Reference in a book ○ Other ○

Rate our site: Amazing

Submit Your Entries Clear Your Entries

Feedback Form

Please fill out this form to help us improve our site.

Name: Jonathan Smith

Comments:

```
Great Site!
```

E-mail Address: ***************

Things you liked:
Site design ☑ Links ☑ Ease of use ☐ Images ☐ Source code ☐

How did you get to our site?:
Search engine ○ Links from another site ● Deitel.com Web site ○ Reference in a book ○ Other ○

Rate our site: Amazing

Submit Your Entries Clear Your Entries

Fig. L.5 Form including radio buttons and drop-down lists. (Part 4 of 4.)

Common Programming Error L.4

When using a group of radio buttons in a form, forgetting to set the **`name`** *attributes to the same name lets the user select all of the radio buttons at the same time, which is a logic error.*

The ***`select`*** element (lines 123–136) provides a drop-down list from which the user can select an item. The **`name`** attribute identifies the drop-down list. The ***`option`*** element (lines 124–135) adds items to the drop-down list. The **`option`** element's ***`selected`*** *attribute* specifies which item initially is displayed as the selected item in the **`select`** element.

L.6 Internal Linking

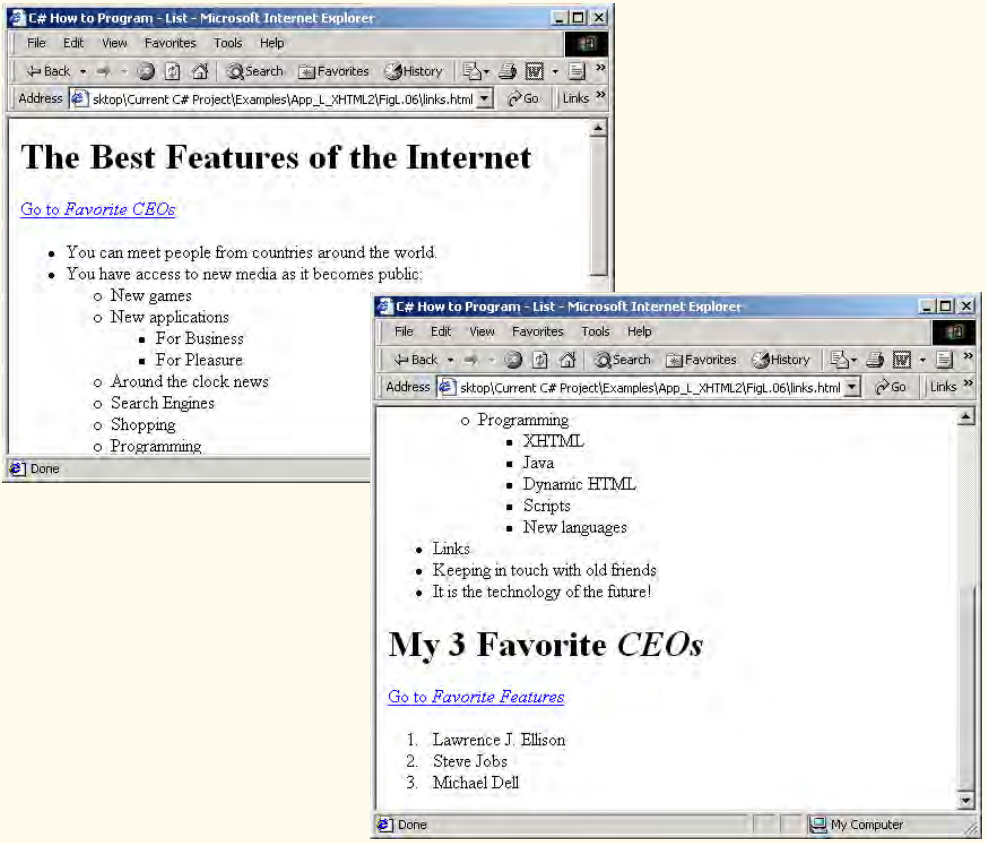
In Appendix K, we discussed how to hyperlink one Web page to another. Figure L.6 introduces *internal linking*—a mechanism that enables the user to jump between locations in the same document. Internal linking is useful for long documents that contain many sections. Clicking an internal link enables users to find a section without scrolling through the entire document.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. L.6: links.html -->
<!-- Internal linking.    -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - List</title>
   </head>

   <body>

      <!-- <a name = ".."></a> creates internal hyperlink -->
      <p><a name = "features"></a></p>

      <h1>The Best Features of the Internet</h1>

      <!-- address of internal link is "#linkname" -->
      <p>
         <a href = "#ceos">Go to <em>Favorite CEOs</em></a>
      </p>

      <ul>
         <li>You can meet people from countries
            around the world.</li>

         <li>You have access to new media as it becomes public:

            <ul>
               <li>New games</li>
               <li>New applications
```

Fig. L.6 Using internal hyperlinks to make pages more easily navigable. (Part 1 of 3.)

```

                  <ul>
                     <li>For Business</li>
                     <li>For Pleasure</li>
                  </ul>

               </li>

               <li>Around the clock news</li>
               <li>Search Engines</li>
               <li>Shopping</li>
               <li>Programming

                  <ul>
                     <li>XHTML</li>
                     <li>Java</li>
                     <li>Dynamic HTML</li>
                     <li>Scripts</li>
                     <li>New languages</li>
                  </ul>

               </li>
            </ul>

         </li>

         <li>Links</li>
         <li>Keeping in touch with old friends</li>
         <li>It is the technology of the future!</li>
      </ul>

      <!-- named anchor -->
      <p><a name = "ceos"></a></p>

      <h1>My 3 Favorite <em>CEOs</em></h1>

      <p>

         <!-- internal hyperlink to features -->
         <a href = "#features">
            Go to <em>Favorite Features</em>
         </a>
      </p>

      <ol>
         <li>Lawrence J. Ellison</li>
         <li>Steve Jobs</li>
         <li>Michael Dell</li>
      </ol>

   </body>
</html>
```

Fig. L.6 Using internal hyperlinks to make pages more easily navigable. (Part 2 of 3.)


C# How to Program - List - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address sktop\Current C# Project\Examples\App_L_XHTML2\FigL.06\links.html
Go Links
Done

The Best Features of the Internet

Go to *Favorite CEOs*

- You can meet people from countries around the world.
- You have access to new media as it becomes public:
  - New games
  - New applications
    - For Business
    - For Pleasure
  - Around the clock news
  - Search Engines
  - Shopping
  - Programming

C# How to Program - List - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address sktop\Current C# Project\Examples\App_L_XHTML2\FigL.06\links.html
Go Links
Done
My Computer

  - Programming
    - XHTML
    - Java
    - Dynamic HTML
    - Scripts
    - New languages
- Links
- Keeping in touch with old friends
- It is the technology of the future!

My 3 Favorite *CEOs*

Go to *Favorite Features*

1. Lawrence J. Ellison
2. Steve Jobs
3. Michael Dell

Fig. L.6 Using internal hyperlinks to make pages more easily navigable. (Part 3 of 3.)

Line 16 contains a *named anchor* (called **`features`**) for an internal hyperlink. To link to this type of anchor inside the same Web page, the href attribute of another anchor element includes the named anchor preceded with a pound sign (as in **`#features`**). Lines 73–74 contain a hyperlink with the anchor **`features`** as its target. Selecting this hyperlink in a Web browser scrolls the browser window to the **`features`** anchor at line 16.

Look-and-Feel Observation L.1

Internal hyperlinks are useful in XHTML documents that contain large amounts of information. Internal links to various sections on the page make it easier for users to navigate the page: They do not have to scroll to find a specific section.

Although not demonstrated in this example, a hyperlink can specify an internal link in another document by specifying the document name followed by a pound sign and the named anchor as in:

```
href = "page.html#name"
```

For example, to link to a named anchor called **`booklist`** in **`books.html`**, **`href`** is assigned **`"books.html#booklist"`**.

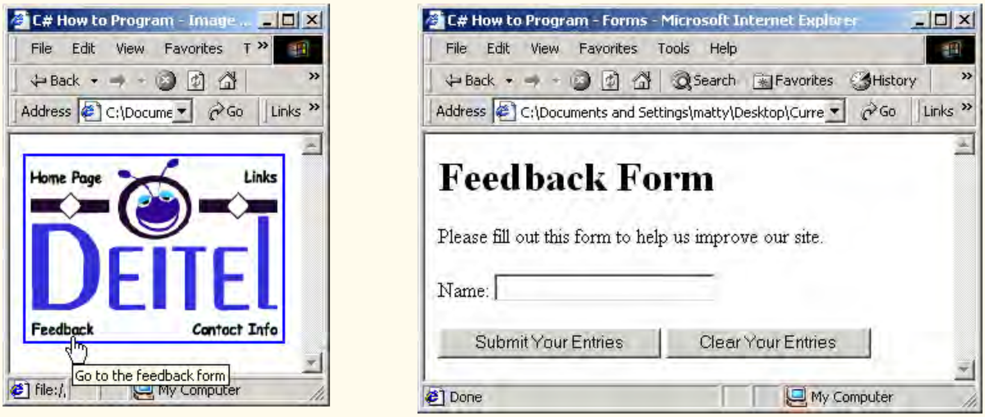
L.7 Creating and Using Image Maps

In Appendix K, we demonstrated how images can be used as hyperlinks to link to other resources on the Internet. In this section, we introduce another technique for image linking called the *image map*, which designates certain areas of an image (called *hotspots*) as links. Figure L.7 introduces image maps and hotspots.

```
<?xml version = "1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. L.7: picture.html         -->
<!-- Creating and using image maps. -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - Image Map
      </title>
   </head>

   <body>

      <p>

         <!-- <map> tag defines image map -->
         <map id = "picture">

            <!-- shape = "rect" indicates rectangular  -->
            <!-- area, with coordinates for upper-left -->
            <!-- and lower-right corners               -->
            <area href = "form.html" shape = "rect"
               coords = "2,123,54,143"
               alt = "Go to the feedback form" />

            <area href = "contact.html" shape = "rect"
               coords = "126,122,198,143"
               alt = "Go to the contact page" />

            <area href = "main.html" shape = "rect"
               coords = "3,7,61,25" alt = "Go to the homepage" />

            <area href = "links.html" shape = "rect"
               coords = "168,5,197,25"
               alt = "Go to the links page" />

            <!-- value "poly" creates hotspot in shape -->
            <!-- of polygon, defined by coords         -->
            <area shape = "poly" alt = "E-mail the Deitels"
              coords = "162,25,154,39,158,54,169,51,183,39,161,26"
               href = "mailto:deitel@deitel.com" />

```

Fig. L.7 Image with links anchored to an image map. (Part 1 of 2.)

```
45               <!-- shape = "circle" indicates a circular -->
46               <!-- area with the given center and radius -->
47               <area href = "mailto:deitel@deitel.com"
48                  shape = "circle" coords = "100,36,33"
49                  alt = "E-mail the Deitels" />
50            </map>
51
52            <!-- <img src =... usemap = "#id"> indicates that -->
53            <!-- specified image map is used with this image  -->
54            <img src = "deitel.gif" width = "200" height = "144"
55               alt = "Deitel logo" usemap = "#picture" />
56         </p>
57
58      </body>
59   </html>
```


C# How to Program - Image ...
File Edit View Favorites T
Back
Address C:\Docume
Go
Links
Home Page
Links
DEITEL
Feedback
Contact Info
Go to the feedback form
file:/
My Computer
C# How to Program - Forms - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address C:\Documents and Settings\matty\Desktop\Curre
Go
Links
Done
My Computer

Feedback Form

Please fill out this form to help us improve our site.

Name:

Submit Your Entries

Clear Your Entries

Fig. L.7 Image with links anchored to an image map. (Part 2 of 2.)

Lines 19–50 define an image map via a ***map*** element. Attribute ***id*** (line 19) identifies the image map. If **id** is omitted, the map cannot be referenced by an image. Shortly, we discuss how to reference an image map. Hotspots are defined with ***area*** elements (as shown on lines 24–26). Attribute **href** (line 24) specifies the link's target (i.e., the resource to which to link). Attributes ***shape*** (line 24) and ***coords*** (line 25) specify the hotspot's shape and coordinates, respectively. Attribute **alt** (line 26) provides alternative text for the link.

Common Programming Error L.5

*Not specifying an **id** attribute for a **map** element prevents an **img** element from using the **map**'s **area** elements to define hotspots.*

The markup on lines 24–26 creates a *rectangular hotspot* (**shape = "rect"**) for the *coordinates* specified in the **coords** attribute. A coordinate pair consists of two numbers representing the location of a point on the *x*-axis and the *y*-axis, respectively. The *x*-axis extends horizontally and the *y*-axis extends vertically from the upper-left corner of the image. Every point on an image has a unique *x*–*y* coordinate. For rectangular hotspots, the required coordinates are those of the upper-left and lower-right corners of the rectangle. In this case, the upper-left corner of the rectangle is located at 2 on the *x*-axis and 123 on the *y*-axis, annotated as *(2, 123)*. The lower-right corner of the rectangle is at *(54, 143)*. Coordinates are measured in pixels.

Common Programming Error L.6

Overlapping coordinates of an image map cause the browser to render the first hotspot it encounters for the area.

The map **`area`** (lines 41–43) assigns the **`shape`** attribute ***`"poly"`*** to create a hotspot in the shape of a polygon, using the coordinates in attribute **`coords`**. These coordinates represent each *vertex*, or corner, of the polygon. The browser connects these points with lines to form the hotspot's area.

The map **`area`** (lines 47–49) assigns the **`shape`** attribute ***`"circle"`*** to create a *circular hotspot*. In this case, the **`coords`** attribute specifies the circle's center coordinates and the circle's radius, in pixels.

To use an image map with an **`img`** element, the **`img`** element's ***`usemap`*** attribute is assigned the **`id`** of a **`map`**. Lines 54–55 reference the image map named **`"picture"`**. The image map is located within the same document, so internal linking is used.

L.8 `meta` Elements

People use search engines to find useful Web sites. Search engines usually catalog sites by following links from page to page and saving identification and classification information for each page. One way that search engines catalog pages is by reading the content in each page's ***`meta`*** elements, which specify information about a document.

Two important attributes of the **`meta`** element are ***`name`***, which identifies the type of **`meta`** element and ***`content`***, which provides the information search engines use to catalog pages. Figure L.8 introduces the **`meta`** element.

Lines 14–16 demonstrate a ***`"keywords"`*** **`meta`** element. The ***`content`*** attribute of such a **`meta`** element provides search engines with a list of words that describe a page. These words are compared with words in search requests. Thus, including **`meta`** elements and their **`content`** information can draw more viewers to your site.

```
 1  <?xml version = "1.0"?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 4
 5  <!-- Fig. L.8: main.html -->
 6  <!-- Using meta tags.    -->
 7
 8  <html xmlns = "http://www.w3.org/1999/xhtml">
 9     <head>
10        <title>C# How to Program - Welcome</title>
11
12        <!-- <meta> tags provide search engines with -->
13        <!-- information used to catalog site        -->
14        <meta name = "keywords" content = "Web page, design,
15           XHTML, tutorial, personal, help, index, form,
16           contact, feedback, list, links, frame, deitel" />
```

Fig. L.8 Using **`meta`** to provide keywords and a description. (Part 1 of 2.)

```
17
18       <meta name = "description" content = "This Web site will
19          help you learn the basics of XHTML and Web page design
20          through the use of interactive examples and
21          instruction." />
22
23    </head>
24
25    <body>
26
27       <h1>Welcome to Our Web Site!</h1>
28
29       <p>
30          We have designed this site to teach about the wonders
31          of <strong><em>XHTML</em></strong>. <em>XHTML</em> is
32          better equipped than <em>HTML</em> to represent complex
33          data on the Internet. <em>XHTML</em> takes advantage of
34          XML’s strict syntax to ensure well-formedness. Soon you
35          will know about many of the great new features of
36          <em>XHTML.</em>
37       </p>
38
39      <p>Have Fun With the Site!</p>
40
41    </body>
42 </html>
```

Fig. L.8 Using `meta` to provide keywords and a description. (Part 2 of 2.)

Lines 18–21 demonstrate a ***`"description"`*** **`meta`** element. The **`content`** attribute of such a **`meta`** element provides a three- to four-line description of a site, written in sentence form. Search engines also use this description to catalog your site and sometimes display this information as part of the search results.

Software Engineering Observation L.1

`meta` *elements are not visible to users and must be placed inside the* ***`head`*** *section of your XHTML document. If* ***`meta`*** *elements are not placed in this section, they will not be read by search engines.*

L.9 `frameset` Element

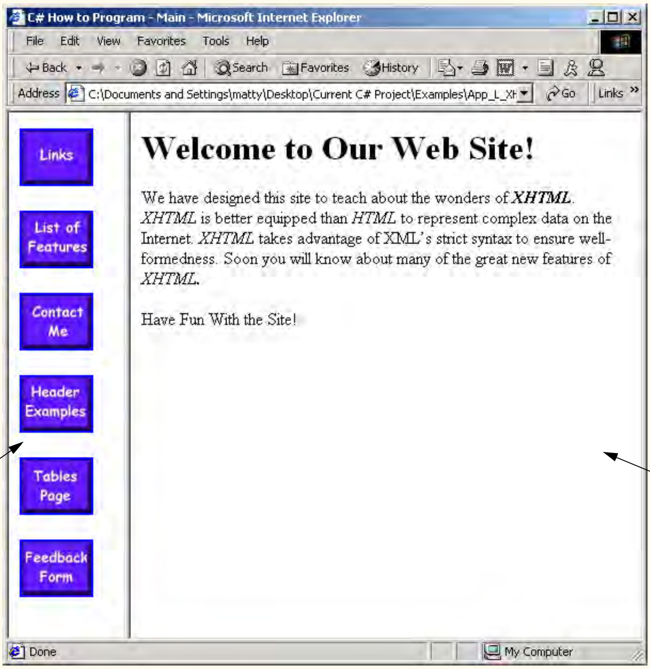
All of the Web pages we have presented in this book have the ability to link to other pages, but can display only one page at a time. Figure L.9 uses *frames*, which allow the browser to display more than one XHTML document simultaneously, to display the documents in Fig. L.8 and Fig. L.10.

Most of our prior examples adhered to the strict XHTML document type. This particular example uses the *frameset* document type—a special XHTML document type specifically for framesets. This new document type is specified in lines 2–3 and is required for documents that define framesets.

A document that defines a frameset normally consists of an **`html`** element that contains a **`head`** element and a ***`frameset`*** element. The ***`<frameset>`*** tag (line 24) informs the browser that the page contains frames. Attribute ***`cols`*** specifies the frameset's column layout. The value of **`cols`** gives the width of each frame, either in pixels or as a percentage of the browser width. In this case, the attribute **`cols = "110,*"`** informs the browser that there are two vertical frames. The first frame extends **`110`** pixels from the left edge of the browser window, and the second frame fills the remainder of the browser width (as indicated by the asterisk). Similarly, **`frameset`** attribute ***`rows`*** can be used to specify the number of rows and the size of each row in a frameset.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!-- Fig. L.9: index.html -->
<!-- XHTML frames I.      -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - Main</title>

      <meta name = "keywords" content = "Webpage, design,
         XHTML, tutorial, personal, help, index, form,
         contact, feedback, list, links, frame, deitel" />

      <meta name = "description" content = "This Web site will
         help you learn the basics of XHTML and Web page design
         through the use of interactive examples
         and instruction." />

   </head>

   <!-- <frameset> tag sets frame dimensions -->
   <frameset cols = "110,*">

      <!-- frame elements specify which pages -->
      <!-- are loaded into given frame        -->
      <frame name = "leftframe" src = "nav.html" />
      <frame name = "main" src = "main.html" />

      <noframes>
         <p>This page uses frames, but your browser does not
         support them.</p>

         <p>Please, <a href = "nav.html">follow this link to
         browse our site without frames</a>.</p>
      </noframes>

   </frameset>
</html>
```

Fig. L.9 Web document containing two frames—navigation and content. (Part 1 of 2.)


C# How to Program - Main - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
Links
List of Features
Contact Me
Header Examples
Tables Page
Feedback Form
Done
My Computer

Address: C:\Documents and Settings\matty\Desktop\Current C# Project\Examples\App_L_XH

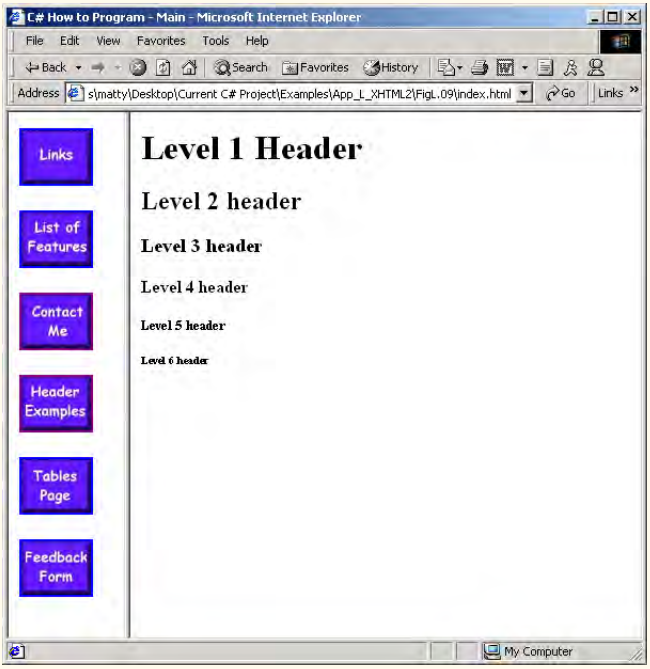
Welcome to Our Web Site!

We have designed this site to teach about the wonders of ***XHTML***. *XHTML* is better equipped than *HTML* to represent complex data on the Internet. *XHTML* takes advantage of XML's strict syntax to ensure well-formedness. Soon you will know about many of the great new features of *XHTML*.

Have Fun With the Site!

Left frame
`leftframe`

Right frame
`main`


C# How to Program - Main - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
Links
List of Features
Contact Me
Header Examples
Tables Page
Feedback Form
My Computer

Address: s\matty\Desktop\Current C# Project\Examples\App_L_XHTML2\FigL.09\index.html

Level 1 Header

Level 2 header

Level 3 header

Level 4 header

Level 5 header

Level 6 header

Fig. L.9 Web document containing two frames—navigation and content. (Part 2 of 2.)

The documents that will be loaded into the **`frameset`** are specified with ***`frame`*** elements (lines 28–29 in this example). Attribute **`src`** specifies the URL of the page to display in the frame. Each frame has **`name`** and **`src`** attributes. The first frame (which covers **`110`** pixels on the left side of the **`frameset`**) is named **`leftframe`** and displays the page **`nav.html`** (Fig. L.10). The second frame is named main and displays the page **`main.html`**.

Attribute **`name`** identifies a frame, enabling hyperlinks in a **`frameset`** to specify the ***`target`*** **`frame`** in which a linked document should display when the user clicks the link. For example,

```
<a href = "links.html" target = "main">
```

loads **`links.html`** in the frame whose **`name`** is **`"main"`**.

Not all browsers support frames. XHTML provides the ***`noframes`*** element (lines 31–37) to enable XHTML document designers to specify alternative content for browsers that do not support frames.

Portability Tip L.1

*Some browsers do not support frames. Use the **`noframes`** element inside a **`frameset`** to direct users to a nonframed version of your site.*

Fig. L.10 is the Web page displayed in the left frame of Fig. L.9. This XHTML document provides the navigation buttons that, when clicked, determine which document is displayed in the right frame.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Fig. L.10: nav.html           -->
<!-- Using images as link anchors. -->

<html xmlns = "http://www.w3.org/1999/xhtml">

   <head>
      <title>C# How to Program - Navigation Bar
      </title>
   </head>

   <body>

      <p>
         <a href = "links.html" target = "main">
            <img src = "buttons/links.jpg" width = "65"
               height = "50" alt = "Links Page" />
         </a><br />

         <a href = "list.html" target = "main">
            <img src = "buttons/list.jpg" width = "65"
               height = "50" alt = "List Example Page" />
         </a><br />
```

Fig. L.10 XHTML document displayed in the left frame of Fig. L.5. (Part 1 of 2.)

```
27
28           <a href = "contact.html" target = "main">
29              <img src = "buttons/contact.jpg" width = "65"
30                 height = "50" alt = "Contact Page" />
31           </a><br />
32
33           <a href = "header.html" target = "main">
34              <img src = "buttons/header.jpg" width = "65"
35                 height = "50" alt = "Header Page" />
36           </a><br />
37
38           <a href = "table1.html" target = "main">
39              <img src = "buttons/table.jpg" width = "65"
40                 height = "50" alt = "Table Page" />
41           </a><br />
42
43           <a href = "form.html" target = "main">
44              <img src = "buttons/form.jpg" width = "65"
45                 height = "50" alt = "Feedback Form" />
46           </a><br />
47        </p>
48
49     </body>
50  </html>
```

Fig. L.10 XHTML document displayed in the left frame of Fig. L.5. (Part 2 of 2.)

Line 29 (Fig. L.9) displays the XHTML page in Fig. L.10. Anchor attribute **`target`** (line 18 in Fig. L.10) specifies that the linked documents are loaded in frame **`main`** (line 30 in Fig. L.9). A **`target`** can be set to a number of preset values: **`"_blank"`** loads the page into a new browser window, **`"_self"`** loads the page into the frame in which the anchor element appears and **`"_top"`** loads the page into the full browser window (i.e., removes the **`frameset`**).

L.10 Nested `frameset`s

You can use the **`frameset`** element to create more complex layouts in a Web page by nesting **`frameset`**s, as in Fig. L.11. The nested **`frameset`** in this example displays the XHTML documents in Fig. L.7, Fig. L.8 and Fig. L.10.

The outer frameset element (lines 23–41) defines two columns. The left frame extends over the first 110 pixels from the left edge of the browser, and the right frame occupies the rest of the window's width. The **`frame`** element on line 24 specifies that the document **`nav.html`** (Fig. L.10) will be displayed in the left column.

Lines 28–31 define a nested **`frameset`** element for the second column of the outer frameset. This **`frameset`** defines two rows. The first row extends 175 pixels from the top of the browser window, and the second occupies the remainder of the browser window's height, as is indicated by **`rows = "175,*"`**. The **`frame`** element at line 29 specifies that the first row of the nested **`frameset`** will display **`picture.html`** (Fig. L.7). The **`frame`** element at line 30 specifies that the second row of the nested **`frameset`** will display **`main.html`** (Fig. L.9).

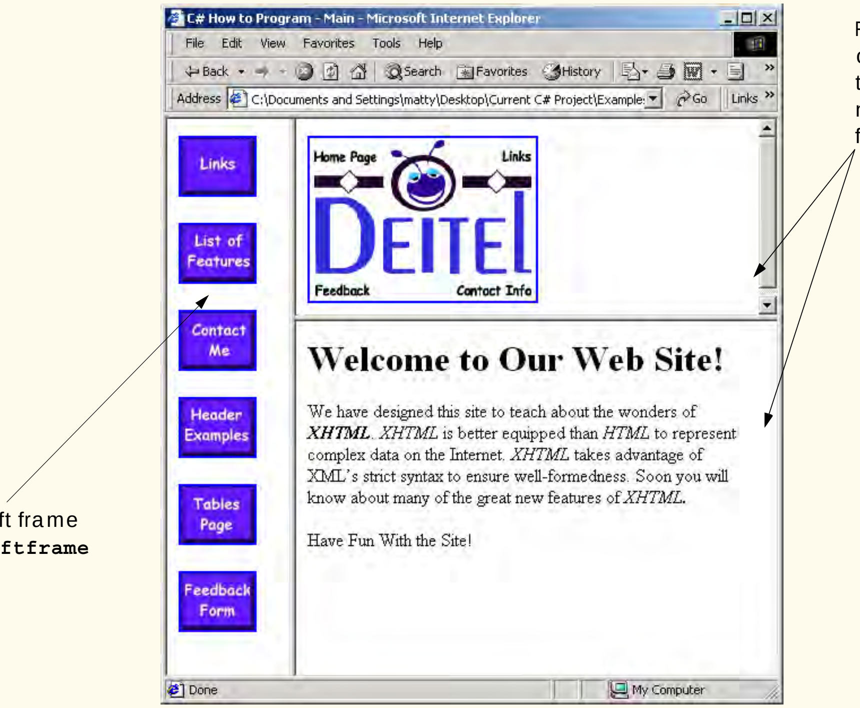
Testing and Debugging Tip L.2

When using nested **`frameset`** *elements, indent every level of* **`<frame>`** *tag. This practice makes the page clearer and easier to debug.*

In this appendix, we presented XHTML for marking up information in tables, creating forms for gathering user input, linking to sections within the same document, using **`<meta>`** tags and creating frames.

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<!-- Fig. L.11: index2.html -->
<!-- XHTML frames II.       -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>C# How to Program - Main</title>

      <meta name = "keywords" content = "Webpage, design,
         XHTML, tutorial, personal, help, index, form,
         contact, feedback, list, links, frame, deitel" />

      <meta name = "description" content = "This Web site will
         help you learn the basics of XHTML and Web page design
         through the use of interactive examples
         and instruction." />

   </head>

   <frameset cols = "110,*">
      <frame name = "leftframe" src = "nav.html" />

      <!-- nested framesets are used to change -->
      <!-- formatting and layout of frameset  -->
      <frameset rows = "175,*">
         <frame name = "picture" src = "picture.html" />
         <frame name = "main" src = "main.html" />
      </frameset>

      <noframes>
         <p>This page uses frames, but your browser does not
         support them.</p>

         <p>Please, <a href = "nav.html">follow this link to
         browse our site without frames</a>.</p>
      </noframes>

   </frameset>
</html>
```

Fig. L.11 Framed Web site with a nested frameset. (Part 1 of 2.)


C# How to Program - Main - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\matty\Desktop\Current C# Project\Example:
Go Links
Links
List of Features
Contact Me
Header Examples
Tables Page
Feedback Form
Home Page Links
DEITEL
Feedback Contact Info
Done
My Computer

Welcome to Our Web Site!

We have designed this site to teach about the wonders of ***XHTML***. *XHTML* is better equipped than *HTML* to represent complex data on the Internet. *XHTML* takes advantage of XML's strict syntax to ensure well-formedness. Soon you will know about many of the great new features of *XHTML*.

Have Fun With the Site!

Right frame contains these two nested frames

Left frame **`leftframe`**

Fig. L.11 Framed Web site with a nested frameset. (Part 2 of 2.)

L.11 Internet and World Wide Web Resources

`courses.e-survey.net.au/xhtml/index.html`
The *Web Page Design—XHTML* site provides descriptions and examples for various XHTML features, such as links, tables, frames and forms. Users can e-mail questions or comments to the Web Page Design support staff.

`www.vbxml.com/xhtml/articles/xhtml_tables`
The *VBXML.com* Web site contains a tutorial on creating XHTML tables.

`www.webreference.com/xml/reference/xhtml.html`
This Web page contains a list of the frequently used XHTML tags, such as header tags, table tags, frame tags and form tags. It also provides a description of each tag.

SUMMARY

- XHTML tables mark up tabular data and are one of the most frequently used features in XHTML.
- The **`table`** element defines an XHTML table. Attribute **`border`** specifies the table's border width, in pixels. Tables without borders set this attribute to **`"0"`**.
- Element **`summary`** summarizes the table's contents and is used by speech devices to make the table more accessible to users with visual impairments.

- Element **`caption`** describe's the table's content. The text inside the **`<caption>`** tag is rendered above the table in most browsers.
- A table can be split into three distinct sections: head (**`thead`**), body (**`tbody`**) and foot (**`tfoot`**). The head section contains such information as table titles and column headers. The table body contains the primary table data. The table foot contains secondary information, such as footnotes.
- Element **`tr`**, or table row, defines individual table rows. Element **`th`** defines a header cell. Text in **`th`** elements usually is centered and displayed in bold by most browsers. This element can be present in any section of the table.
- Data within a row are defined with **`td`**, or table data, elements.
- Element **`colgroup`** groups and formats columns. Each **`col`** element can format any number of columns (specified with the **`span`** attribute).
- The document author has the ability to merge data cells with the **`rowspan`** and **`colspan`** attributes. The values assigned to these attributes specify the number of rows or columns occupied by the cell. These attributes can be placed inside any data-cell tag.
- XHTML provides forms for collecting information from users. Forms contain visual components, such as buttons that users click. Forms may also contain nonvisual components, called hidden inputs, which are used to store any data, such as e-mail addresses and XHTML document file names used for linking.
- A form begins with the **`form`** element. Attribute **`method`** specifies how the form's data is sent to the Web server.
- The **`"text"`** input inserts a textbox into the form. Textboxes allow the user to input data.
- The **`input`** element's **`size`** attribute specifies the number of characters visible in the **`input`** element. Optional attribute **`maxlength`** limits the number of characters input into a textbox.
- The **`"submit"`** input submits the data entered in the form to the Web server for processing. Most Web browsers create a button that submits the form data when clicked. The **`"reset"`** input allows a user to reset all **`form`** elements to their default values.
- The **`textarea`** element inserts a multiline textbox, called a textarea, into a form. The number of rows in the textarea is specified with the **`rows`** attribute, the number of columns (i.e., characters) with the **`cols`** attribute.
- The **`"password"`** input inserts a password box into a form. A password box allows users to enter sensitive information, such as credit card numbers and passwords, by "masking" the information input with another character. Asterisks are the masking character used for most password boxes. The actual value input is sent to the Web server, not the asterisks that mask the input.
- The checkbox input allows the user to make a selection. When the checkbox is selected, a check mark appears in the checkbox. Otherwise, the checkbox is empty. Checkboxes can be used individually and in groups. Checkboxes that are part of the same group have the same **`name`**.
- A radio button is similar in function and use to a checkbox, except that only one radio button in a group can be selected at any time. All radio buttons in a group have the same **`name`** attribute value, but different attribute **`value`**s.
- The **`select`** input provides a drop-down list of items. The **`name`** attribute identifies the drop-down list. The **`option`** element adds items to the drop-down list. The **`selected`** attribute, like the **`checked`** attribute for radio buttons and checkboxes, specifies which list item is displayed initially.
- Image maps designate certain sections of an image as links. These links are more properly called hotspots.

- Image maps are defined with `map` elements. Attribute `id` identifies the image map. Hotspots are defined with the `area` element. Attribute `href` specifies the link's target. Attributes `shape` and `coords` specify the hotspot's shape and coordinates, respectively, and `alt` provides alternative text.
- One way that search engines catalog pages is by reading the `meta` elements's contents. Two important attributes of the `meta` element are `name`, which identifies the type of `meta` element, and *`content`*, which provides information a search engine uses to catalog a page.
- Frames allow the browser to display more than one XHTML document simultaneously. The `frameset` element informs the browser that the page contains frames. Not all browsers support frames. XHTML provides the `noframes` element to specify alternative content for browsers that do not support frames.

TERMINOLOGY

`action` attribute
`area` element
`border` attribute
browser request
`<caption>` tag
checkbox
`checked` attribute
`col` element
`colgroup` element
`cols` attribute
`colspan` attribute
`coords` element
form
`form` element
`frame` element
`frameset` element
header cell
`hidden input` element
hotspot
`href` attribute
image map
`img` element
`input` element
internal hyperlink
internal linking
`map` element
`maxlength` attribute
`meta` element
`method` attribute
`name` attribute
navigational frame
nested `frameset` element
nested tag
`noframes` element
password box
`"radio"` (attribute value)
`rows` attribute (`textarea`)
`rowspan` attribute (`tr`)
`selected` attribute
`size` attribute (`input`)
`table` element
`target = "_blank"`
`target = "_self"`
`target = "_top"`
`tbody` element
`td` element
textarea
`textarea` element
`tfoot` (table foot) element
`<thead>...</thead>`
`tr` (table row) element
`type` attribute
`usemap` attribute
`valign` attribute (`th`)
`value` attribute
Web server
XHTML form
x–*y* coordinates

SELF-REVIEW EXERCISES

L.1 State whether the following statements are *true* or *false*. If *false*, explain why.

a) The width of all data cells in a table must be the same.
b) Framesets can be nested.
c) You are limited to a maximum of 100 internal links per page.
d) All browsers can render `frameset`s.

L.2 Fill in the blanks in each of the following statements:

a) Assigning attribute **`type`** __________ in an **`input`** element inserts a button that, when clicked, clears the contents of the form.

b) The layout of a **`frameset`** is set by including the __________ attribute or the __________ attribute inside the **`<frameset>`** tag.

c) The __________ element marks up a table row.

d) __________ are usually used as the masking characters in a password box.

e) The common shapes used in image maps are __________, __________ and __________.

L.3 Write XHTML markup to accomplish each of the following tasks:

a) Insert a framed Web page, with the first frame extending 300 pixels across the page from the left side.

b) Insert a table with a border of 8.

c) Indicate alternative content to a **`frameset`**.

d) Insert an image map in a page, using **`deitel.gif`** as an image and **`map`** with **`name = "hello"`** as the image map, and set the **`alt`** text to “**`hello`**”.

ANSWERS TO SELF-REVIEW EXERCISES

L.1 a) False. You can specify the width of any column, either in pixels or as a percentage of the table width. b) True. c) False. You can have an unlimited number of internal links. d) False. Some browsers are unable to render a **`frameset`** and must therefore rely on the information that you include inside the **`<noframes>`**…**`</noframes>`** tags.

L.2 a) **`"reset"`**. b) **`cols`**, **`rows`**. c) **`tr`**. d) asterisks. e) **`poly`** (polygons), **`circle`**s, **`rect`** (rectangles).

L.3 a) `<frameset cols = "300,*">`…`</frameset>`

b) `<table border = "8">`…`</table>`

c) `<noframes>`…`</noframes>`

d) `<img src = "deitel.gif" alt = "hello" usemap = "#hello" />`

EXERCISES

L.4 Categorize each of the following as an element or an attribute:

a) **`width`**

b) **`td`**

c) **`th`**

d) **`frame`**

e) **`name`**

f) **`select`**

g) **`type`**

L.5 What will the **`frameset`** produced by the following code look like? Assume that the pages referenced are blank with white backgrounds and that the dimensions of the screen are 800 by 600. Sketch the layout, approximating the dimensions.

```
<frameset rows = "20%,*">
   <frame src = "hello.html" name = "hello" />
      <frameset cols = "150,*">
         <frame src = "nav.html" name = "nav" />
         <frame src = "deitel.html" name = "deitel" />
      </frameset>
</frameset>
```

L.6 Write the XHTML markup to create a frame with a table of contents on the left side of the window, and have each entry in the table of contents use internal linking to scroll down the document frame to the appropriate subsection.

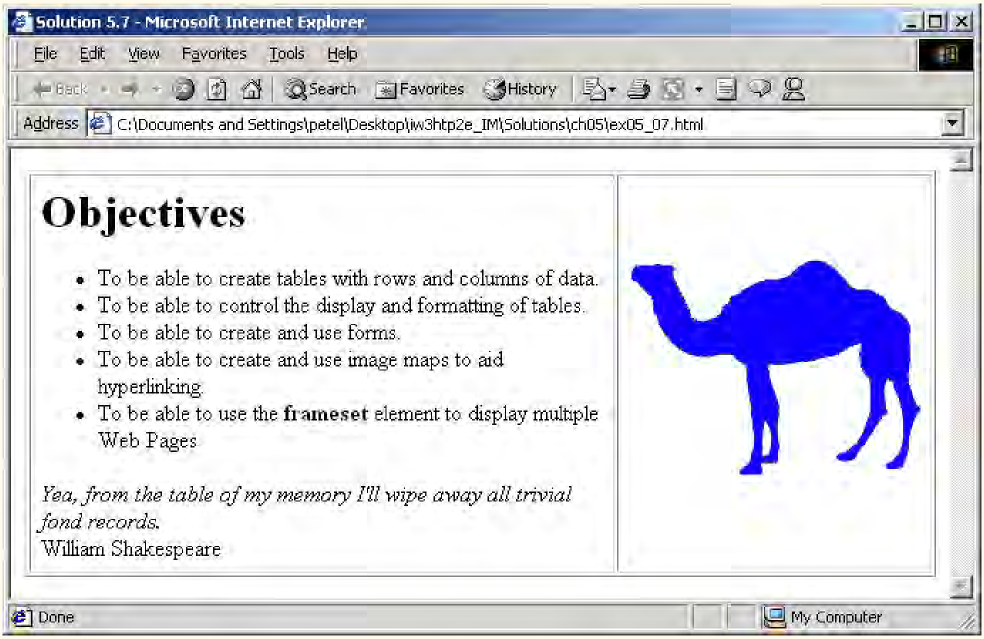
L.7 Create XHTML markup that produces the table shown in Fig. L.12. Use `<em>` and `<strong>` tags as necessary. The image (`camel.gif`) is included in the Appendix L examples directory on the CD-ROM that accompanies this book.

L.8 Write an XHTML document that produces the table shown in Fig. L.13.

L.9 A local university has asked you to create an XHTML document that allows potential students to provide feedback about their campus visit. Your XHTML document should contain a form with textboxes for a name, address and e-mail. Provide checkboxes that allow prospective students to indicate what they liked most about the campus. These chec boxes should include students, location, campus, atmosphere, dorm rooms and sports. Also, provide radio buttons that ask the prospective students how they became interested in the university. Options should include friends, television, Internet and other. In addition, provide a textarea for additional comments, a submit button and a reset button.

L.10 Create an XHTML document titled "How to Get Good Grades." Use `<meta>` tags to include a series of keywords that describe your document.

L.11 Create an XHTML document that displays a tic-tac-toe table with player X winning. Use `<h2>` to mark up both Xs and Os. Center the letters in each cell horizontally. Title the game, using an `<h1>` tag. This title should span all three columns. Set the table border to 1.

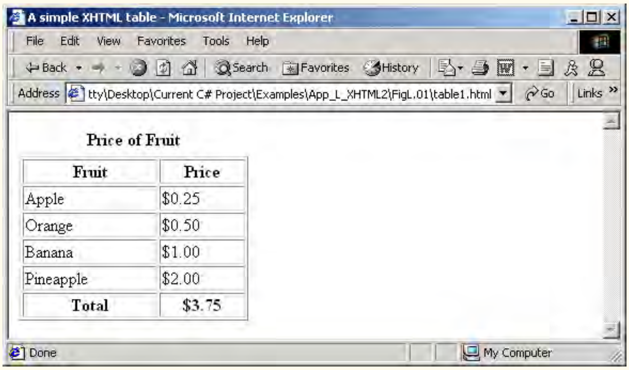

Solution 5.7 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address C:\Documents and Settings\petel\Desktop\iw3htp2e_IM\Solutions\ch05\ex05_07.html
Done
My Computer

Objectives

- To be able to create tables with rows and columns of data.
- To be able to control the display and formatting of tables.
- To be able to create and use forms.
- To be able to create and use image maps to aid hyperlinking.
- To be able to use the **frameset** element to display multiple Web Pages

Yea, from the table of my memory I'll wipe away all trivial fond records.
William Shakespeare

Fig. L.12 XHTML table for Exercise L.7.


A simple XHTML table - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address tty\Desktop\Current C# Project\Examples\App_L_XHTML2\FigL.01\table1.html
Go Links
Done
My Computer

Price of Fruit

Fruit	Price
Apple	$0.25
Orange	$0.50
Banana	$1.00
Pineapple	$2.00
Total	**$3.75**

Fig. L.13 XHTML table for Exercise L.8.

Bibliography

Albahari, B., P. Drayton and B. Merrill, *C# Essentials*. Cambridge, MA: O'Reilly & Associates, 2001.

Anderson, R., A. Homer, R. Howard and D. Sussman. *A Preview of Active Server Pages+*. Birmingham, UK: Wrox Press, 2001.

Anderson, R., B. Francis, A. Homer, R. Howard, D. Sussman and K. Watson, *ASP .NET*. Chicago, IL: Wrox Press, Inc., 2001.

Archer, T. *Inside C#*. Redmond, WA: Microsoft Press, 2001.

Blaha, M. R., W. J. Premerlani and J. E. Rumbaugh. "Relational Database Design Using an Object-Oriented Methodology." *Communications of the ACM*, Vol. 31, No. 4, April 1988, 414–427.

Carr, D. "Hitting a High Note." *Internet World*. March 2001, 71.

Carr, D. "Slippery SOAP." *Internet World*. March 2001, 72–74.

Carr, D. F. "Dave Winer: The President of Userland and SOAP Co-Creator Surveys the Changing Scene." *Internet World*. March 2001, 53–58.

Chappel, D. "Coming Soon: The Biggest Platform Ever." *Application Development Trends Magazine*, May 2001,15.

Chappel, D. "A Standard for Web Services: SOAP vs. ebXML." *Application Development Trends*, February 2001, 17.

Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, June 1970.

Codd, E. F. "Further Normalization of the Data Base Relational Model." *Courant Computer Science Symposia*, Vol. 6, *Data Base Systems*. Upper Saddle River, N.J.: Prentice Hall, 1972.

Codd, E. F. "Fatal Flaws in SQL." *Datamation*, Vol. 34, No. 16, August 15, 1988, 45–48.

Conard, J., P. Dengler, B. Francis, J. Glynn, B. Harvey, B. Hollis, R. Ramachandran, J. Schenken, S. Short and C. Ullman. *Introducing .NET*. Birmingham, UK: Wrox Press, 2000.

Correia, E. J. "Visual Studio .NET to Speak in Tongues." *Software Development Times*, April 2001, 12.

Cornell, G. and J. Morrison. *Moving to VB .NET: Strategies, Concepts, and Code*. Berkeley, CA: Apress Publishing, 2001.

Date, C. J. *An Introduction to Database Systems, Seventh Edition*. Reading, MA: Addison-Wesley Publishing, 2000.

Davydov, M. "The Road to the Future of Web Services." *Intelligent Enterprise*. May 2001, 50–52.

Deitel, H. M. and Deitel, P. J. *Java How To Program, Fourth Edition*. Upper Saddle River, NJ: Prentice Hall, 2001

Deitel, H. M., Deitel, P. J. and T. R. Nieto. *Visual Basic 6 How To Program*. Upper Saddle River, NJ: Prentice Hall, 1999.

Deitel, H. M., P. J. Deitel, T. R. Nieto, T. M. Lin and P. Sadhu. *XML How To Program*. Upper Saddle River, NJ: Prentice Hall, 2001

Deitel, H. M. *Operating Systems, Second Edition*. Reading, MA: Addison Wesley Publishing, 1990.

Dejong, J. "Raising the Bar." *Software Development Times*, March 2001, 29–30.

Dejong, J. "Microsoft's Clout Drives Web Services." *Software Development Times*, March 2001, 29, 31.

Dejong, J. "One-Stop Shopping: A Favored Method." *Software Development Times*, February 2001, 20.

Erlanger. L. ".NET Services." *Internet World*, March 2001, 47.

Erlanger. L. "Dissecting .NET." *Internet World*, March 2001, 30–36.

Esposito, D. "Data Grid In-Place Editing." *MSDN Magazine*, June 2001, 37–45.

Esposito, D. "Server-Side ASP .NET Data Binding: Part 2: Customizing the Data Grid Control." *MSDN Magazine*, April 2001, 33–45.

Finlay, D. "UDDI Works on Classification, Taxonomy Issues." *Software Development Times*, March 2001, 3.

Finlay, D. "New York Prepares for .NET Conference." *Software Development Times*, June 2001, 23.

Finlay, D. "GoXML Native Database Clusters Data, Reduces Seek Time." *Software Development Times*, March 2001, 5.

Fontana, J. "What You Get in .NET." *Network World*, April 2001, 75.

Galli, P. and R. Holland. ".NET Taking Shape, But Developers Still Wary." *eWeek*, June 2001, pages 9, 13.

Gillen, A. "Sun's Answer to .NET." *EntMag*, March 2001, 38.

Gillen, A. "What a Year It's Been." *EntMag*, December 2000, 54.

Gladwin, L. C. "Microsoft, eBay Strike Web Services Deal." *Computer World*, March 2001, 22.

Grimes, R. "Make COM Programming a Breeze with New Feature in Visual Studio .NET." *MSDN Magazine*, April 2001, 48–62.

Gunnerson, E. *A Programmer's Introduction to C#: Second Edition*. New York, NY: Apress, 2001.

Harvey, B., S. Robinson, J. Templeman and K. Watson. *C# Programming With the Public Beta*. Birmingham, UK: Wrox Press, 2000.

Holland, R. "Microsoft Scales Back VB Changes." *eWeek*, April 2001, 16.

Holland, R. "Tools Case Transition to .NET Platform." *eWeek*, March 2001, 21.

Hulme, G, V. "XML Specification May Ease PKI Integration." *Information Week*, December 2000, 38.

Hutchinson, J. "Can't Fit Another Byte." *Network Computing*, March 2001, 14.

Jepson, B. “Applying .NET to Web Services.” *Web Techniques*, May 2001, 49–54.

Jones, B. Sams Teach Yourselft C# in 21 Days. Indianapolis, IN: Sams Publishing, 2002.

Karney. J. “.NET Devices.” *Internet World*, March 2001, 49–50.

Kiely, D. “Doing .NET In Internet Time.” *Information Week*, December 2000, 137–138, 142–144, 148.

Kirtland, M. “The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework.” *MSDN Magazine*, September 2000 **<msdn.microsoft.com/msdnmag/issues/0900/WebPlatform/WebPlatform.asp>**.

Levitt, J. “Plug-And-Play Redefined.” *Information Week*, April 2001, 63–68.

McCright, J. S. and D. Callaghan. “Lotus Pushes Domino Services.” *eWeek*, June 2001, 14.

Michaelis, M. and P. Spokas. *C# Developer’s Headstart*. New York, NY: Osbourne/McGraw-Hill, 2001.

“Microsoft Chimes in with New C Sharp Programming Language.” Xephon Web site. June 30, 2000 **<www.xephon.com/news/00063019.html>**.

Microsoft Corporation, *Microsoft C# Language Specifications*. Redmond, VA: Microsoft Press, 2001.

Microsoft Developer Network Documentation. Visual Studio .NET CD-ROM, 2001.

Microsoft Developer Network Library. .NET Framework SDK. Microsoft Web site <**msdn.microsoft.com/library/default.asp**>.

Moran, B. “Questions, Answers, and Tips.” *SQL Server Magazine*, April 2001, 19–20.

MySQL Manual. MySQL Web site <**www.mysql.com/doc/**>.

Oracle Technology Network Documentation. Oracle Web site. <**otn.oracle.com/docs/content.html**>.

Otey, M. “Me Too .NET.” *SQL Server Magazine*. April 2001, 7.

Papa, J. “Revisiting the Ad-Hoc Data Display Web Application.” *MSDN Magazine*, June 2001, 27–33.

Powell, R. and R. Weeks. C# and the .NET Framework: The C# Perspective. Indianapolis, IN: Sams Publishing, 2002.

Pratschner, S. “Simplifying Deployment and Solving DLL Hell with the .NET Framework.” *MSDN Library*, September 2000 **<msdn.microsoft.com/library/techart/dplywithnet.htm>**.

Prosise, J. “Wicked Code.” *MSDN Magazine*, April 2001, 121–127.

Relational Technology, *INGRES Overview*. Alameda, CA: Relational Technology, 1988.

Ricadela, A. and P. McDougall. “eBay Deal Helps Microsoft Sell .NET Strategy.” *Information Week*, March 2001, 33.

Ricadela, A. “IBM Readies XML Middleware.” *Information Week*, December 2000, 155.

Richter, J. “An Introduction to Delegates.” *MSDN Magazine*, April 2001, 107–111.

Richter, J. “Delegates, Part 2." *MSDN Magazine*, June 2001, 133–139.

Rizzo, T. “Let’s Talk Web Services.” *Internet World*, April 2001, 4–5.

Rizzo, T. “Moving to Square One.” *Internet World*, March 2001, 4–5.

Robinson, S., O. Cornes, J. Glynn, B. Harvey, C. McQueen, J. Moemeka, C. Nagel, M. Skinner and K. Watson. *Professional C#*. Birmingham, UK: Wrox Press, 2001.

Rollman, R. "XML Q & A." *SQL Server Magazine*, April 2001, 57–58.

Rubinstein, D. "Suit Settled, Acrimony Remains." *Software Development Times*, February 2001, pages 1, 8.

Rubinstein, D. "Play It Again, XML." *Software Development Times*, March 2001, 12.

Scott, G. "Adjusting to Adversity." *EntMag*, March 2001, 38.

Scott, G. "Putting on the Breaks." *EntMag*, December 2000, 54.

Sells, C. "Managed Extensions Bring .NET CLR Support to C++." *MSDN Magazine*. July 2001, 115–122.

Seltzer, L. "Standards and .NET." *Internet World*, March 2001, 75–76.

Shohoud, Y. "Tracing, Logging, and Threading Made Easy with .NET." *MSDN Magazine*, July 2001, 60–72.

Sliwa, C. "Microsoft Backs Off Changes to VB .NET." *Computer World*, April 2001, 14.

Songini, Marc. "Despite Tough Times, Novell Users Remain Upbeat." *Computer World*, March 2001, 22.

Spencer, K. "Cleaning House." *SQL Server Magazine*, April 2001, 61–62.

Spencer, K. "Windows Forms in Visual Basic .NET." *MSDN Magazine*, April 2001, 25–45.

Stonebraker, M. "Operating System Support for Database Management." *Communications of the ACM*, Vol. 24, No. 7, July 1981, 412–418.

Surveyor. J. ".NET Framework." *Internet World*, March 2001, 43–44.

Tapang, C. C. "New Definition Languages Expose Your COM Objects to SOAP Clients." *MSDN Magazine*, April 2001, 85–89.

Thai, T. and H. Q. Lam. *.NET Framework*. Cambride, MA: O'Reilly & Associates, Inc., 2001.

Troelsen, A. *C# and the .NET Platform*. New York, NY: Apress, 2001.

Utley, C. *A Programmer's Introduction to Visual Basic .NET*. Indianapolis, IN: Sams Publishing, 2001.

Visual Studio .NET ADO .NET Overview. Microsoft Developers Network Web site <**msdn.microsoft.com/vstudio/nextgen/technology/adoplusdefault.asp**>.

Ward, K. "Microsoft Attempts to Demystify .NET." *EntMag*, December 2000, 1.

Waymire, R. "Answers from Microsoft." *SQL Server Magazine*, April 2001, 71–72.

Whitney, R. "XML for Analysis." *SQL Server Magazine*, April 2001, 63–66.

Wille, C. *Presenting C#*. Indianapolis, IN: Sams Publishing, 2000.

Winston, A. "A Distributed Database Primer." *UNIX World*, April 1988, 54–63.

Zeichick, A. "Microsoft Serious About Web Services." *Software Development Times*, March 2001, 3.

Index

Symbols

`!` (logical NOT) 160, 163
`!=` is not equal to 81, 160
`"` (double quotation) 64, 67
`""` 844
`%` (modulus operator) 76, 77, 78
`%=` (modulus assignment operator) 121
`&` (bitwise AND) 1496, 1497, 1500
`&` (boolean logical AND) 160, 166
`&` (menu access shortcut) 522, 523
`&&` (logical AND) 160, 162
`&=` (bitwise AND assignment operator) 1507
`&amp;` 1387
`&copy;` 1387
`&frac14;` 1446
`&lt;` 1445, 1446
(GCD) greatest common divisor 235
(GUI) graphical user interface 23, 37, 69, 475
* SQL wildcard character 881, 908
`*/` end a multiline comment 61
`*=` (multiplication assignment operator) 121
`+` operator 78, 647
`++`, preincrement/postincrement 121
`+=` (addition assignment operator) 120
`--`, predecrement/postdecrement 121, 122
`.` (dot operator) 181
`.disco` file extension 1057
`.vsdisco` file extension 1057
`//` single-line comment 61
`/=` (division assignment operator) 121
`;` (empty statement) 70, 104
`;` (statement terminator) 64
`<` is-less-than operator 81
`<%@Page…%>` directive 954
`<%@Register…%>` directive 1015
`<<` (left-shift operator) 1496, 1497, 1498, 1504
`<<=` (left-shift assignment operator) 1508
`<=` less than or equal 81, 160
`<>` angle brackets 840
`<?` and `?>` delimiters 878
`=` (assignment operator) 74, 120
`-=` (subtraction assignment operator) 121
`==` comparison operator 81, 638
`>` is-greater-than operator 81
`>=` is-greater-than-or-equal-to operator 81
`>>` (right-shift operator) 1496, 1497, 1504
`>>=` (right-shift assignment operator) 1508
? regular expression metacharacter 766
? SQL wildcard character 908
`?:` (ternary conditional operator) 101, 123
`[]` (brackets) 238, 239, 249
\ separator character 761
`\'` escape sequence 68
`\\` escape sequence 68
`\n` escape sequence 67, 68
`\r` escape sequence 68
`\t` escape sequence 68
`\u`*yyyy* unicode format 1356
`^` (bitwise exclusive OR) 1496, 1497, 1500, 1508
`^` (boolean logical exclusive OR) 160
`^=` (bitwise exclusive OR assignment operator) 1507
_ (underscore) 62
`{` (left brace) 63, 104
`|` (bitwise inclusive OR) 1496, 1497, 1500, 1508
`|` (boolean logical inclusive OR) 160
`|=` (bitwise inclusive OR assignment operator) 1507
`||` (logical OR) 160, 161
`}` (right brace) 63, 104
`~` (bitwise complement operator) 1496, 1497, 1500
`,` (comma) 149

A

A 12-element array 238
A binary search tree containing 12 values 1169
a element 1381, 1385, 1439, 1444
A graphical representation of a binary tree 1169
A graphical representation of the **InsertAtBack** operation 1158
A graphical representation of the **InsertAtFront** operation 1158
A graphical representation of the **RemoveFromBack** operation 1160
A graphical representation of the **RemoveFromFront** operation 1159
A portion of a **Shape** class hierarchy 347
A property of structure **Color** 689
abbreviating an assignment expression 120
Abort method of class **Thread** 593
AbortRetryIgnore method 148
AboutBox method of class **AxMediaPlayer** 736
Abs method of class **Math** 182
absolute positioning 963
absolute value 182
abstract base class 392
abstract class 392, 393, 403, 404, 417
Abstract class **Employee** definition 404
abstract data type (ADT) 22, 282
abstract derived class 404
abstract method 392, 403, 404
Abstract **Shape** base class 394
abstraction 344
AcceptButton property 478
AcceptSocket method of class **TcpListener** 1109
AcceptsReturn property 489
access method 285
access shared data 607
access shortcut 522
accessibility 1217, 1249, 1251, 1258, 1262, 1263, 1264
accessibility aids in Visual Studio .NET 1217, 1218
Accessibility Wizard 1251, 1254, 1258
Accessibility Wizard initialization option 1252
Accessibility Wizard mouse cursor adjustment tool 1254
AccessibilityDescription property of class **Control** 1230
AccessibilityName property of class **Control** 1230
AccessibleDescription property of class **Control** 1225
AccessibleName property of class **Control** 1225
AccessibleRole enumeration 1226
AccessibleRole property of class **Control** 1226
Accessing and displaying a database's data 921
Accessing **private** class members from client code generates syntax errors 291
accessing shared memory without synchronization 601
action 64, 326
action attribute 1247, 1465
action oriented 282
action symbol 98
action/decision model of programming 100
Activation property of class **ListView** 553
Active Accessibility 1249
Active Server Pages (ASP) .NET 16, 26
active window 477
ActiveLinkColor property of class **LinkLabel** 531
ActiveMdiChild property of class **Form** 567
ActiveX 1363
ActiveX COM control integration in Visual Basic .NET 1365
ActiveX control 28, 1364
ActiveX DLL 28
Acts designed to ensure Internet access for people with disabilities 1214
Ad attribute 974
Ada programming language 10, 591
add custom control to a form 583
Add member of enumeration **MenuMerge** 568
Add method 722, 873
Add method of class **ArrayList** 1066, 1189, 1193
Add method of class **Hashtable** 1091, 1204
Add method of class **TreeNodeCollection** 549
Add method of **Columns** collection 1013
Add Reference dialog 70
Add Reference dialog DLL Selection 1368
Add Tab menu item 560
Add User Control... option in Visual Studio .NET 580
Add Web Reference dialog 1055
Add Windows Form... option in Visual Studio .NET 567
Adding a new label to the form 48
adding a proxy class for a Web service 1053
Adding a reference to an assembly in Visual Studio .NET 70
adding a Web service reference to a project 1055
adding **Web References** in Visual Studio 1053
addition assignment operator (**+=**) 120
Addition program to add two values entered by the user 72
Addition.cs 72
AddLine method of class **GraphicsPath** 715
AddressBook.cs 930
Adjusting up window element size 1253
"administrative" section of the computer 4
ADO .NET 25, 897, 920, 941
AdRotator class 971, 972
AdRotator class demonstrated on a Web form 972
AdRotator.aspx 972
AdRotator.aspx.cs 973
AdRotatorInformation.xml 974
ADT (abstract data type) 22, 282
advanced accessibility settings in Microsoft Internet Explorer 5.5 1263
Advanced Research Projects Agency (ARPA) 13
Advantage Hiring, Inc. 1297

`AdvertisementFile` property of class `AdRotator` 972
`AdvertisementFile` used in `AdRotator` example 974
advertisment 987
`AfterSelect` event of class `TreeView` 549
742
aggregation 290
Airline Reservation Web service 1076
airline reservation Web service 1076
airline Web Service in design view 1078
algebraic notation 77
algorithm 21, 95, 106, 222
algorithm for traversing a maze 279
`Alignment` property 49
allocating arrays 240
allocating memory 242
Allowed implicit conversions 194
`AllowPaging` property 1027
`AllowPaging` property of a `DataGrid` control 1027
`AllowSorting` property of windows form designer 1027
Alphabetic icon 42
alphabetizing 638
`alt` attribute 27, 1217, 1383, 1442
Alt key 511, 522
`Alt` property 514
`<alt>` tag 1263
ALU (arthimetic and logic unit) 4, 1496
America's Job Bank 1295
American Society for Female Entrepreneurs 1297
American Standard Code for Information Interchange (ASCII) 27
ampersand (`&`) 1387
699
`Analysis.cs` 118
Analytical Engine mechanical computing device 11
ancestor node 847
anchor 1380, 1385
anchor control 486
`Anchor` property 487
anchoring a control 486
Anchoring demonstration 486
`AND` 918, 919
`And` method of class `BitArray` 1508
AND operator (boolean logical) 160
AND operator (logical) 160, 162
angle bracket (`<>`) 840, 1375
animated character 24
animation 687
720
`answer` element 1248
Apache 1405
API (application programming interface) 685
APL progamming language 19
`Append` method of class `StringBuilder` 654, 655
`Append` methods of class `StringBuilder` 655
`AppendFormat` method of class `StringBuilder` 655, 657
`AppendText` method of class `File` 761
Apple Computer, Inc. 5, 1351
`Application` class 529
application class definition 108
application programming interface (API) 685
application service provider (ASP) 1297
Application that writes an XML representation of a `DataSet` to a file 939
application tracing 1027
Application with accessibility features 1226
`Application.Exit` method 529, 539
`ApplicationException` class 448
`Aquent.com` 1300
arc angle 704
`arc` method 704
705
`area` element 1416, 1477
ARGB values 688
argument 64, 181, 185
`ArgumentOutOfRangeException` 636, 647, 658
`ArgumentOutOfRangeException` class 1193
arithmetic and logic unit (ALU) 4, 1496
Arithmetic assignment operators 120
arithmetic calculation 76
arithmetic mean (average) 78
arithmetic operator 76, 77
arithmetic overflow 439, 440
ARPA (Advanced Research Projects Agency) 13
ARPAnet 14
`ArrangeIcons` value in `LayoutMdi` enumeration 570
array 22, 237
array allocated with `new` 249
array as an object 239
array automatically initialized to zeros 249
array bounds 250
`Array` class 1185
array elements of primitive data types passed call-by-value 250
array indexer (`[]`) 238
array of a nonprimitive data type 240
array of arrays 266
array of primitive data types 240
array subscript operator, `[]` 1193
`ArrayList` class 722, 1066, 1067, 1185, 1188, 1193
`ArrayListTest.cs` 1189
`ArrayReferenceTest.vb` 254
arrays are C# objects 281
arrays as object 253
arrays passed call-by-reference 250
`article.xml 839`
`article.xml` displayed by Internet Explorer 842
`ASC` (ascending order) 259, 882, 909, 911
ascent 698
ASCII (American Standard Code for Information Interchange) 27, 1351
ASCII character, test for 514
ASCX code for the header 1015
ASCX file 1015
ASMX file 1042
ASMX file rendered in Internet Explorer 1042
ASMX page 1044
ASP (Active Server Pages) .NET 16
ASP .NET comment 954
ASP .NET server controls 949
ASP.NET Web service project type 1052
`ASP.NET_SessionId` cookie 996

ASPX file 949
`.aspx` file extension 949
ASPX file for the guest book application 1007
ASPX file that allows a user to select an author from a drop-down list 1021
ASPX file that presents a list of programming languages 989
ASPX file that takes ticket information 1078
ASPX listing for the guest book page 1007
ASPX page that displays book information 994
ASPX page that displays the Web server's time 953
ASPX page with tracing turned off 1028
assembler 7
assembly 69, 332
assembly language 6
Assembly `TimeLibrary` contains class `Time3` 329
Assembly `TimeLibrary` used from class `AssemblyTest` 333
`<assign>` tag 1242
`assign` element 1248
assign value to a variable 74
Assigning derived-class references to base-class references 387
assignment operator (`=`) 74, 81, 120
assignment statement 74
associate left to right 85, 123
associate right to left 78, 85, 115, 123
associativity of operators 78, 123, 239
asterisk (`*`) 905, 1480
asterisk (`*`) indicating multiplication 76
asterisk (`*`) occurence indicator 866
asynchronous event 440
`ATTLIST` element 867
attribute 843, 1381
attribute of an element 1433
attributes 125
audio clip 591
audio-video interleave (AVI) 733
Aural Style Sheet 1264
AuralCSS 1250
`authorISBN` table of `books` database 898, 899, 900
`authorization` 1021
`Authors` table of `books` database 898
`authors` table of `books` database 898
`Authors.aspx` 1021
`Authors.aspx.cs` 1023
auto hide 40
auto hide, disable 40
`AutoEventWireup` attribute of ASP.NET page 954
automatic duration 212
automatic garbage collection 592
Autos window 1317
`AutoScroll` property 495
average 78
average calculation 106, 109
`Average1.cs` 107
`Average2.cs` 112
AVI (audio-video interleave) 733
`AxAgent` control 742, 749
`AxMediaPlayer` class 735

B

`B` property of structure `Color` 690
B2B 1041
Babbage, Charles 11
`BackColor` property 47, 485
background color 47, 518
`BackgroundImage` property 485
backslash (`\`) 67
Ballmer, Steve 20
bandwidth 14, 718
 772
bar chart 243
base 2 number system 1496
base case(s) 216, 220, 221
base class 125, 343, 344, 346
base-class constructor 348
base-class constructor call syntax 365
base-class default constructor 348
 772
base-class method is overridden in a derived class 366
base-class `Private` member 347
base-class reference 388, 389
baseline 1462
BASIC (Beginner's All-Purpose Symbolic Instruction Code) 8
batch 5
batch processing 4
BCPL programming language 7
Beginner's All-Purpose Symbolic Instruction Code (BASIC) 8
behavior 125, 281
Berkeley System Distribution (BSD) 18
Berners-Lee, Tim 14
`BilingualJobs.com` 1297
binary 176
binary digit 757
binary number system 1496
binary operator 74, 76, 115, 163
binary representation 1496, 1498
binary search 260, 261, 262, 278
Binary search of a sorted array 262
binary search tree 1169, 1175, 1176
binary tree 1146, 1169, 1176
binary tree sort 1176
`BinaryFormatter` class 760, 782
`BinaryReader` class 807, 1118
`BinarySearch` method of class `Array` 1187
`BinarySearch` method of class `ArrayList` 1194
`BinaryTreeLibrary.cs` 1170, 1178
`BinaryWriter` class 1118
`Bind` method of class `Socket` 1108
binding a server to a port 1108
bit (size of unit) 757, 1351, 1496
bit manipulation 757, 1496
bit mask 1498
bit representation of an integer 1498
bit set 1508
`BitArray` class 1185, 1496, 1508, 1510
`BitArray` indexer 1508
`BitArrayTest.cs` 1508
`Bitmap` class 714
`BitOperations.cs` 1501
bits-and-bytes level 1496
`BitShift.cs` 1505
bitwise AND (`&`) 1496, 1497, 1500
bitwise AND assignment operator (`&=`) 1507
bitwise assignment operator 1507
bitwise complement operator (`~`) 1496, 1497, 1500
bitwise exclusive OR (`^`) 1496, 1497, 1500, 1508

bitwise exclusive OR assignment operator (`^=`) 1507
bitwise inclusive OR (`|`) 1496, 1497, 1500, 1508
bitwise inclusive OR assignment operator (`|=`) 1507
bitwise operators 501, 1496, 1497, 1498, 1507
BizTalk 25, 844
BizTalk Framework 885
BizTalk markup using an offer Schema 885
BizTalk Schema Library 885
BizTalk Server (BTS) 885
BizTalk Terminologies 885
`Black` `static` property of structure `Color` 689
Blackjack 1063
Blackjack game that uses `Blackjack` Web service 1067
`Blackjack` Web service 1063
`Blackjack.cs` 1067
`BlackjackService.asmx.cs` 1063
`Blackvoices.com` 1296
blank line 62
block 104, 187, 212, 1109
`block` element 1242, 1247
block for input/output 595
block scope 213, 290
blocked state 594
`Blue` `static` property of structure `Color` 689
Bluetooth 1299
body 63
`body` element 1015, 1378, 1433, 1434
body of a class definition 63, 284
body of a loop 146
body of the `if` structure 80
body of the method definition 63
body of the `while` 105
body section 1433
Bohm, C. 97, 170
`Bold` member of enumeration `FontStyle` 697
`Bold` property of class `Font` 696
`book.xdr` 871
`book.xml` 870, 872
`book.xsd` 872
`books` database 898
`books` database table relationships 904
`bookxdrfail.xml` 876
`bookxsd.xml` 872
`bookxsdfail.xml` 876
`bool` primitive data type 100, 196
`bool` promotions 194
`bool` values 166
`bool` variables initialized to `false` 212
boolean decision 1508
boolean flag 1508
boolean logical AND operator (`&`) 160
boolean logical exclusive OR operator (`^`) 160
boolean logical inclusive OR operator (`|`) 160
`border = "0"` 1400
`border` attribute 1400, 1459
`Boss` class inherits from class `Employee` 405
bottom tier 952
BounceKeys 1256
boundary of control 579
bounding rectangle 702, 704
bounding rectangle for an oval 704
`br` (line break) element 1386, 1445
braces (`{` and `}`) 104, 114, 143, 158, 186
braces not required in `case`s 155
braces that delimit a compound statement 104
braces, matching left and right 75
brackets (`[]`) 238, 239, 249
braille display 1217, 1250
braille keyboard 1249
`Brassringcampus.com` 1302
Break All button 1318
break program 1318
`break` statement 21, 152, 158
`<break>` tag 1242
breakpoint 1313
breakpoint condition 1320
breakpoint hit count 1320
breakpoint, active 1319
breakpoint, disabled 1319
breakpoint, set 1314
Breakpoints window 1319
`BreakTest.cs` 158
brick-and-mortar store 987
browser request 1405, 1462
`Brush` class 685, 690, 701
BSD (Berkeley System Distribution) 18
BTS (BizTalk Server) 885
bubble sort 257
Bubble sort using delegates 426
Bubble-sort `Form` application 427
`BubbleSortForm.cs` 427
bubbling a small value 259
buffer 601
buffer empty 601
`BufferedStream` class 760
buffering 25
Build menu 38, 52
Build Solution option 52
building-block approach 12, 95, 140
built-in array capabilities 1185
built-in data type 73, 326
Business letter DTD 866
business logic 953
business rule 953
business-to-business (B2B) 1041
button 34, 69, 70, 71, 476
`Button` class 488
button label 488
`Button` properties and events 490
Buttons for message dialogs 148
byte 758, 1496
byte offset 795
`byte` primitive data type 196, 1496
`byte` promotions 194

C

`C` formatting code 151
C programming language 7, 1376, 1431
C-style comment 61
C# generating exception for invalid subscripting 249
C# How to Program 3
C# keywords 98
C# Language Specification 449
C# primitive data types 196
C# program for examination-results problem 118
C# programming language 9, 19, 844
C#'s view of a file of *n* bytes 760
 760
C#'s view of an n-byte file 760
C++ programming language 8, 1376
C++-style single-line comment 61
cache 920
Calculating compound interest with `for` 149
Calculating factorials with a recursive method 217
calculation 4
`call` element 1248

Call Stack 457, 1322
call-by-reference 250
call-by-value 250
called method 180
caller 180
callerID attribute 1248
calling method 180
CallXML 27, 1243
callxml element 1243, 1248
CallXML example that reads three ISBN values 1245
CallXML **hangup** element 1243
CampusCareerCenter.com 1302
CancelButton property 478
Candidate Recommendation 15
capacity of a collection 1188
Capacity property of class **ArrayList** 1188
Capacity property of class **ArrayList** 1194
Capacity property of class **StringBuilder** 652
caption element 1234, 1459
Card class 664
Card dealing and shuffling simulation 665
card games 664
Card.cs 664
Career.com 1295
CareerPath.com 1295
CareerWeb 1295
carriage return 68
carry bit 1284
Cascade value in **LayoutMdi** enumeration 570
Cascading Style Sheets (CSS) 15, 27
case-sensitive language 62, 105, 1375
case statement 152, 154, 155
cast 92
CAST eReader 1231
cast operation 388
cast operator 112, 115, 194
catch all exception types 442
catch block (or handler) 442, 446, 450
Categorized icon 42
CCircle5 class inherits from class **CPoint3** and overrides a finalizer method 374
CDATA flag 867
CD-ROM 3
CDT>Circle class contains an *x-y* coordinate and a radius 351
Ceil method of class **Math** 182
Center for Applied Special Technology 1231, 1265
Centering the text in the label 50
central processing unit (CPU) 4
CERN (the European Organization for Nuclear Research) 14
CGI (Common Gateway Interface) 1405
CGI script 1405, 1465
chance 200
changes in server-side data 1405
Changing a property in the code view editor 128
Changing a property value at run time 130
Changing property **BackColor** 47
Char array 636
char primitive data type 196
char promotions 194
Char structure 633, 661
char variable 73
Char.IsDigit method 661
Char.IsLetter method 661
Char.IsLetterOrDigit method 661
Char.IsLower method 663
Char.IsPunctuation method 663
Char.IsSymbol method 663
Char.IsUpper method 663
Char.IsWhiteSpace method 663
Char.ToLower method 663
Char.ToUpper method 663
Char’s **static** character-testing methods and case-conversion methods 661
character 758, 1352
character class 669
Character classes 669
character constant 634
character entity reference 28
character set 73, 92, 758, 1351
character string 64
Characters property of class **AxAgent** 748
Charles Goldfarb 887
CharMethods.cs 661
checkbox 476, 488, 498, 1406, 1466
CheckBox class 498
checkbox label 498
CheckBox properties and events 498
CheckBoxes property of class **ListView** 553
CheckBoxes property of class **TreeView** 548
CheckBoxTest.cs 499
checked attribute 1409, 1469
checked context 466
checked operator 466
Checked property 498, 502
Checked property of class **MenuItem** 524
Checked property of class **TreeNode** 549
CheckedChanged event 499, 502
CheckedIndices property of class **CheckedListBox** 540
CheckedItems property of class **CheckedListBox** 540
CheckedListBox and a **ListBox** used in a program to display a user selection 541
CheckedListBox and **ListBox** used in a program to display a user selection 541
CheckedListBox class 521, 534, 539
CheckedListBox properties, methods and events 540
CheckedListBoxTest.cs 541
CheckState property 498
CheckStateChanged event 499
Chemical Markup Language (CML) 25
725
ChessGame.cs 725
723
ChiefMonster™ 1301
Child class for MDI demonstration 574
child element 841, 843, 1169
child node 547, 847
child window minimized 568
Child.cs 574
choice element of **form** tag 1241
choice element of **menu** tag 1241

`<choice>` tag 1242
circle 1450
`Circle` class contains an *x-y* coordinate and a radius 351
`Circle` class marked up with XML comments 1333
`Circle` class that inherits from class `Point` 385
`Circle2` class that inherits from class `Point` 355
`Circle2` class that inherits from class `Point2` 397
`Circle3` class that inherits from class `Point2` 359
`Circle3` class that inherits from class `Point3` 421
`Circle3.cs 421`
`Circle4` class that inherits from class `Point3` but does not use `protected` data 364
`Circle5` class inherits from class `Point3` and overrides a finalizer method 374
`CircleTest` class marked up with XML comments 1336
`CircleTest` demonstrates class `Circle` functionality 354
`CircleTest3` demonstrates class `Circle3` functionality 360
`CircleTest4` demonstrates class `Circle4` functionality 366
circular buffer 616, 627
circular hotspot 1418, 1478
`CircularBuffer.cs` 617
circumference 92
`CityWeather.cs` 1085
clarity 2
class 11, 22, 62, 179, 282
Class average program with counter-controlled repetition 107
Class `Control` layout properties 487
Class `Control` properties and methods 485
class definition 62, 71, 284, 1338
Class `FrmInheritance`, which inherits from class `Form`, contains a button (**Learn More**) 575
Class `FrmVisualTest`, which inherits from class `VisualForm.FrmInheritance`, contains an additional button 577
class hierarchy 383, 392
`class` keyword 62, 124
class library 13, 344
class name 62
class scope 213, 290
class-scope variable hidden by method-scope variable 290
class that stores equation information 1093
class that stores weather information about a city 1085
Class View (Visual Studio .NET) 333
"class-wide" information 312
Class-average program with sentinel-controlled repetition 112
Classes that derive from class `Brush` 690
classes to implement abstract data types 327
`clear` element 1248
`Clear` method of class `Array` 1188
`Clear` method of class `ArrayList` 1189
`Clear` method of class `TreeNodeCollection` 549
`Clear` method of `DataSet` 930
`Clear` method of `Hashtable` 1204
`clearDigits` element 1248, 1249
click a button 488
`Click` event of class `MenuItem` 523, 525
Clicker 4 1261
clicking the mouse 34, 70
client 6, 326
client interacting with server and Web server. Step 2: The HTTP response, `HTTP/1.0 200 OK` 951
Client portion of a client/server stream-socket connection 1114
client tier 953
`Client.cs` 1087, 1114, 1132
client/server chat 1111
client/server computing 6
`ClipRectangle` property of class `PaintEventArgs` 578, 579
clock 580
`ClockUserControl.cs` 580
close a file 788
Close button icon 53
`Close` method of class `Form` 478
`Close` method of class `Socket` 1109, 1119
`Close` method of class `StreamReader` 788
`Close` method of class `TcpClient` 1110, 1111
closed polygon 708
`CloseFigure` method of class `GraphicsPath` 715
closing a project 38
closing brace (`}`) 114
CLR (Common Language Runtime) 18
CLS (Common Language Specification) 18
CML (Chemical Markup Language) 25
`CNN.com` 987
COBOL (COmmon Business Oriented Language) 10, 19, 1431
Code generated by the IDE for `lblWelcome` 127
code reuse 1185
code value 1352
code-behind file 949, 1015
Code-behind file for a page that displays the Web server's time 955
Code-behind file for a page that updates the time every minute 955
Code-behind file for page demonstrating the `AdRotator` class 973
Code-behind file for the guest book application 1009
Code-behind file for the log in page for authors application 1016
Code-behind file for the page that allows a user to choose an author 1023
code-behind file for the reservation page 1079
Code-behind file for the word generator page 979

Code-behind file for `WebForm1.aspx` generated by Visual Studio .NET 964
code-behind file in Web services 1042
Code-behind file that writes cookies to the client 991
coercion of arguments 193
coin-tossing 201, 234
`col` element 1403, 1460
`colgroup` element 1402, 1403, 1460
`Collapse` method of class `TreeNode` 549
collapse node 547
collapsed code 125
collapsing a tree 39
`Collect` method of `GC` 316, 377
collection 1185
collection class 1185
`Collegegrads.com` 1302
collision detection 722
`Color 689`
color constant 689
color manipulation 687
`Color` methods and properties 689
`Color` property of class `ColorDialog` 693
`Color` structure 686, 688
`Color` structure members 689
`ColorDialog` class 693
`ColorDialog` used to change background and text color 693
`cols` attribute 1408, 1421, 1465, 1480
`colspan` attribute 1403, 1460
column 897, 898, 1398
column heading 1400
column number 906
`Columns` collection 1013
COM (Component Object Model) 28, 1362, 1364
COM component 28, 1363
COM limitation 1363
`ComboBox` class 476, 521, 542
`ComboBox` demonstration 543
`ComboBox` properties and events 543
`ComboBox` used to draw a selected shape 544
`ComboBoxTest.cs` 544
comma (`,`) 149
comma in a `for` structure header 149
comma-separated list of arguments 73, 74, 75, 144, 181, 186
command-and-control system 591
command prompt 60, 64
`Commands` property of interface 749
comment 83
comment (`//`) 61, 73, 1377, 1432
comment at the end of each line 83
`Comment` constant of enumeration `XmlNodeType` 849
comments in HTML 1377
commercial application 757
`CommissionWorker` class inherits from class `Employee` 406
COmmon Business Oriented Language (COBOL) 10
Common Form properties and events 478
Common Gateway Interface (CGI) 1405, 1463
Common Language Runtime (CLR) 18, 449, 463
Common Language Specification (CLS) 18
Common Programming Error 12
Common Runtime Library proxy 1364
Commonly used `Math` class methods 182
`CompareTo` method of `IComparable` 1178
`CompareTo` method of structure `Char` 664
compilation error 1312
compile-time error 61
compile, debug setting 1314
compiler 7
Compiler error messages generated from overloaded methods 225
compiler optimization 402
compiling 1146
complete representation of a program 110, 116
complex curve 715
complex number class 430
complex type 873
complexity theory 222
`ComplexNumber.cs` 430
`complexType` element 873
component 8, 477
Component Object Model (COM) 28, 1362, 1364
Component Pascal progamming language 19
Components and controls for Windows Forms 477
composition 22, 290, 306, 344, 346
compound statement 104, 114
comprehensive job sites 1290
computation 3
computational complexity 259
computer 3
computer program 3
computer programmer 3
Computing the sum of the elements of an array 242
`Concat` method of class `String` 647
`Concat Shared` method 648
`Concat static` method 648
concatenation of `string`s 83
concrete class 392
concurrency 591
concurrent producer and consumer threads 607
concurrent programming 591
concurrent threads 607
condition 80
condition variable 614
conditional AND 162
conditional expression 101, 286
conditional operator (`?:`) 101, 123, 160
conditional OR operator 162
`conference` element 1249
Confusing the equality operator `==` with the assignment operator `=` 81
`Connect` method of class `TcpListener` 1110
connect to a database 921
connected lines 708
connection 1108
connection attempt 1110
connection between client and server terminates 1111
connection to a server 1110, 1138
Connection`Connection` property of `OleDbCommand` 928
connectionless service 1108
connectionless transmission with datagrams 1120, 1140

connection-oriented, streams-based transmission 1120, 1140
connector symbol 98
consistent state 289, 298
console application 60
`Console` class 64, 760
console window 60, 66, 606, 615
Console windows with tabs and without tabs 1224
`Console.ReadLine` method 73
`Console.Write` method 66
`Console.WriteLine` method 64
`const` and `readonly` class member demonstration 318
`const` keyword 22, 242, 317
`const` variable 156
constant 317
constant integral value 152
constant variable 156, 242
constituent controls 579
constrained version of a linked list 1160
constructor 286, 292, 377, 1338
consume method 601
consumer 615
consumer thread 601
consuming a Web service 1046
`contact.html` 1381, 1386, 1440, 1445
container 476, 478
Container class for chess pieces 723
container elements 841
`Contains` method of class `ArrayList` 1189, 1194
`Contains` method of class `Stack` 1198
`ContainsKey` method of `Hashtable` 1204
`content` attribute of `meta` element 871, 955, 1478
context-sensitive help 43
contiguous memory location 237
`continue` statement in a `for` structure 21, 158, 159
`ContinueTest.cs` 159
control 20, 36, 475, 477
control boundary 579
`Control` class 484, 579
control layout 38
control layout and properties 484
`Control` property 514
control structure 97, 99, 140, 171
control structures in iteration 222
control structures in recursion 222
control variable 140, 142, 144
control variable final value 140
control variable inital value 140
control variable name 140
controlling expression 152
controls 40
control-structure nesting 99
control-structure stacking 99, 166
`ControlToValidate` property of class `RegularExpressionValidator` 979
converge on a base case 216
`Convert` class of namespace `System` 193
Converting a binary number to decimal 1281
Converting a hexadecimal number to decimal 1282
converting an integral value to a floating-point value 195
Converting an octal number to decimal 1281
cookie 987, 988, 989, 996
  deletion 988
  domain 997
  expiration 988
  expiration date 988
  `Expires` property 988
  header 988
`CookieContainer` class 1066, 1067
Cookies being read from a client in an ASP .NET application 995
Cookies getting created in an ASP .NET Web application 991
`Cookies` property of `Request` class 996
Cookies recieved in an ASP .NET Web application 995
`Cooljobs.com` 1303
coordinate system 686, 687
coordinates (0, 0) 686
`coords` element 1418, 1477
`Copy` method of class `Array` 1185
`Copy` method of class `File` 761
copy of an argument 197
copyright 28
`CopyTo` method of class `String` 636
CORDA Technologies 1217
corporate culture 1293, 1296
correctness of input values 250
`Cos` method of class `Math` 182
cosine 182
`count` attribute if `prompt` element 1241
`Count` property of class `ArrayList` 1194
`Count` property of `Hashtable` 1204
counter 106, 108, 110
counter-controlled repetition 106, 107, 113, 116, 140, 142, 222
counter variable 152
Counter-controlled repetition with the `for` structure 142
Counter-controlled repetition with `while` structure 141
`Counts` property of `HttpSessionState` class 1003, 1006
CPU (Central Processing Unit) 4
craps program 207
Create and write to a sequential-access file 777
`Create` method of class `File` 761
create new classes from existing class definitions 284
`CreateDirectory` method of class `Directory` 762
`CreateInstance` method of class `Array` 1188
`CreateRandomAccessFile.cs` 799
`CreateSequentialAccessFile.cs` 777
`CreateText` method of class `File` 761
creating a child form to be added to an MDI form 567
Creating a new Windows application 45
Creating a `Panel` with scrollbars 496
Creating an ASP.NET Web Application in Visual Studio 960
creating comment Web pages 1339
creating data types 327
Creating files for random-access file-processing applications 799
Credit-inquiry program 788
`CreditInquiry.cs` 788
Crystal Decisions 1513

Crystal Reports 1513
Crystal Reports Designer 1515
CSS (Cascading Style Sheets) 15, 27, 1235
CSS2 1235
Ctrl key 511
`Current` property of `IEnumerator` 1194
current time 580
`CurrentPageIndex` property 1027
`CurrentPageIndex` property of a `DataGrid` control 1027
`CurrentThread static Thread` property 595
`CurrentValue` event of class `CheckedListBox` 541
Curriculum progamming language 19
cursor 64, 66
curve 715
custom control 578, 579
Custom control added to a `Form` 583
Custom control added to the `ToolBox` 582
Custom control added to the **ToolBox** 582
Custom control creation 579, 582
Custom-control creation 582
Examples
  Custom-control creation 582
customization 988
customize an application 40
Customize Toolbox 1364
Customize Toolbox dialog selecting an ActiveX control 1364
Customize Toolbox... option in Visual Studio .NET 581
customize Visual Studio .NET 35, 38
customizing the **Toolbox** 733
`CustomValidator` class 1015, 1020
`Cyan static` property of structure `Color` 689
cylinder 398
`Cylinder` class inherits from class `Circle4` and overrides method Area 368
`Cylinder` class inherits from class `Circle4` and overrides method `Area` 368
`Cylinder2` class inherits from class `Circle2` 398
`Cylinder3` class inherits class `Circle3` 423
`Cylinder3.cs` 423

D

`D` formatting code 151
dangling-else problem 103
`DarkBlue static` property of structure `Color` 697
`DarkGray static` property of structure `Color` 689
`Dash` member of enumeration `DashStyle` 715
`DashCap` enumeration 715
`DashCap` property of class `Pen` 715
`DashStyle` enumeration 715
`DashStyle` property of class `Pen` 715
data 3
data abstraction 326
data entry 69
data hiding 285
Data hierarchy 759
data hierarchy 758
data in support of actions 326
data independence 16
data manipulation 717
data member 282, 292
Data menu 38
data representation of an abstract data type 326
data structure 22, 26, 237, 285, 1146
data tier 952
data type 73
database 896, 1375, 1431
Database information input into a DataGrid 1023
database management system (DBMS) 759, 896
database table 897
datagram 1120
`DataGrid` class 921, 928
`DataGrid` control 1013, 1027
`DataSet` class 920, 938
`DataSource` property of a `DataGrid` control 1013
`DataTable` class 1013
`data-type` attribute 881
date and time 580
`Date` class encapsulates day, month and year information 306
`DateTime` structure 580
DBCS (double byte character set) 1353
DBMS (database management system) 759
DB2 897
Dead thread state 593
deadlock 600, 601
debug configuration setting 1314
Debug menu 38, 1315
Debug sample program 1313
Debug toolbar 1318
`DebugClass.cs` 1325
`DebugExample.cs` 1313
debugger 1313
debugging 1027, 1312, 1377, 1432
Debugging methods 1322
debugging, begin 1315
debugging, call stack 1322
debugging, step into 1323
debugging, step out 1323
debugging, step over 1318
decendant node 847
decimal digit 758
`decimal` primitive data type 149, 196
`decimal` promotions 194
decision symbol 80, 98, 100
`DeckOfCards.cs` 665
declaration 72, 105, 186
declaration space of a class 213
declare each variable on a separate line 73
declaring arrays 240
decreasing order 257
decrement expression 144
decrement of loop 140
decrement operator, `--` 121, 122
default constructor 293, 348
default font 697
default namespace 846, 847
Default namespaces demonstration 846
default package 288
default properties 126
default setting 1258
default sorting order is ascending 909
`default` statement 152
`defaultnamespace.xml 846`, 846
definite repetition 107

Definitions of **TreeNode** and **Tree** for a binary search tree 1170
degree 704
deitel:BooksType 873
deitel@deitel.com 3
del element 1446
delegate 425
Delegate class 426
delegate reference 483
delegate, create new 484
DELETE FROM 905, 919
Delete method of class **Directory** 762
Delete method of class **File** 761
DELETE statement 919
DeleteCommand property of **OleDbAdapter** 927
DeleteDialog.cs 830
deletion 1149
delimit 840
Demomstrating the **ArrayList** class 1189
Demonstrating keyboard events 513
Demonstrating order in which constructors and destructors are called 376
demonstrating **ref** and **out** parameters 198, 452
Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators 1501
Demonstrating the linked list 1155
Demonstrating the logical operators 164
Demonstrating window auto-hide 41
Demonstrating XML namespaces 844
Demonstration of methods that draw lines, rectangles and ellipses 702
dequeue operation of queue 327, 1165
derived class 125, 343, 344, 346
DESC 909, 910
descending order 259
Description column 1339
Description property of a **WebMethod** attribute 1052
Description property of a **WebService** attribute 1051
deselected state 501
Deserialize method of class **BinaryFormatter** 782
design mode 52, 53
Design mode of Web Form designer 962
design units 696
design view of a Web service 1054
destructor 312
destructor method 312
Details of **Click** event 483
diacritic 1352
dialog 36, 68, 70
diameter 92
diamond symbol 98, 100, 105, 145, 155, 177
dice game 207
dice-rolling program 244
Dice.com 1299
digit 73
direct-access files 794
direct base class 343
directive in ASP .NET page 954
762
Directory class methods (partial list) 762
DirectoryInfo class 559, 761
DISCO (Discovery file) 1054, 1447, 1450
.disco file extension 1057
disconnected 920
discovery (DISCO) files 1054, 1447, 1450
discovery file 1054
disk 3, 13
disk I/O completion 440
disk space 1148
dismiss (hide) a dialog 70
Display Color Settings 1253
Display member of enumeration **GraphicsUnit** 696
display output 68
Display Settings 1251
displaying a line of text 64
displaying a phrase 61
displaying data on the screen 85
Displaying multiple lines in a dialog 68
displaying numeric data 243
displaying numeric data graphically 243
Displaying the bit representation of an integer 1498
DisplayLogoForm.cs 718
DisplayQueryResults.cs 928
distributed computing 6, 1041
diversity 1297
divide-and-conquer approach 21, 179, 182
divide by zero 111, 446
DivideByZeroException class 446
DivideByZeroTest.cs 443
division assignment operator (**/=**) 121
division by zero is undefined 327
D-link 1217
DLL (dynamic link library) 22, 1362, 1364, 1368, 1369
.dll file 1368
"DLL hell" 1362
DNS (domain name server) 950
DNS lookup 950
do/while flowchart 157
do/while repetition structure 21, 98, 143, 156, 167, 171
dock control 486
Dock property 487
docking a control 486
docking demonstration 487
DockPadding property 487
document 61
Document member of enumeration **GraphicsUnit** 696
Document Object Model (DOM) 847
Document Style and Semantics Specification Language (DSSSL) 16
document type 954, 1376
Document Type Definition (DTD) 865, 866, 868
Dogfriendly.com 1303
DOM (Document Object Model) 847
DOM parser 847
DOM structure of an XML document illustrated by a class 851
domain name server (DNS) 950
Domain property of **HttpCookie** class 997
dot (**.**) operator 70, 181, 290, 313, 347
double 73

double-byte character set (DBCS) 1353
`Double` class 464
double-clicking 34
`double` primitive data type 112, 114, 197
`double` promotions 194
double quotes (`""`) 64, 67, 844
double-selection structure 98, 117, 168, 170
double-subscripted array 265, 266
Double-subscripted array with three rows and four columns 266
`DoWhileLoop.cs` 156
down-arrow 38, 47
drag 42
drag and drop 479
draw on control 579
draw shapes 685
`DrawArc` method of class `Graphics` 705
`DrawArcs.cs` 705
`DrawEllipse` method of class `Graphics` 547, 702
drawing a line 701
drawing a rectangle 701
drawing an oval 701
`DrawLine` method of class `Graphics` 702, 708
`DrawPie` method of class `Graphics` 547, 705
`DrawPolygon` method of class `Graphics` 708
`DrawPolygons.cs` 708
`DrawRectangle` method of class `Graphics` 547, 701, 702
`DrawShapes.cs` 712
`DrawStarsForm.cs` 715
`DrawString` method of Class `Graphics` 693
`DrawString` method of class `Graphics` 697
drop-down list 476, 542
`DropDown` style for `ComboBox` 544
`DropDownList` class 1015
`DropDownList` style for `ComboBox` 544
`DropDownStyle` property of class `ComboBox` 543, 544
DSSSL (Document Style and Semantics Specification Language) 16
DTD (Document Type Definition) 865, 868
`.dtd` file extension 868
DTD for a business letter 866
DTD repository 871
dummy value 109
duplicate elimination 1177
duplicate of datagram 1120
duration 212
dynamic content 9
dynamic data structures 1146
dynamic help 42
Dynamic Help window 42, 44
dynamic link library (`.dll`) 69
dynamic link library (DLL) 22, 329, 332, 1362, 1364, 1368, 1369
dynamic memory allocation 1148, 1149

E

`E` formatting code 151
e-mail (electronic mail) 14
EagleEyes 1250
EBNF (Extended Backus-Naur Form) grammar 866
echo a packet back to a client 1120
ECMA (European Computer Manufacturer's Association) 18
ECMAScript 977
Edit menu 38
Edit menu in Internet Explorer 69
editable list 544
efficient (Unicode design basis) 1351
Eiffel progamming language 19
eights position 1277
`eLance.com` 1300
electronic devices with two stable states 757
element 237, 873, 1375
`Element` constant of enumeration `XmlNodeType` 848
`!ELEMENT` element 866
element of chance 200
element type declaration 866
elements 840
`ElementType` 871
eliminate resource leaks 450
Ellipse bounded by a rectangle 704
`else` statement 101
`eltOnly` attribute 871
emacs text editor 1376, 1431
Emacspeak 1217
e-mail (electronic mail) 14, 1109, 1441
e-mail anchor 1382, 1441
embedded parentheses 77
employee 390
`Employee` class encapsulates employee name, birthday and hire date 308
`EmployeesTest` class tests the `Employee` class hierarchy 412
empty `case` 154
empty element 844, 1443, 1445
`EMPTY` keyword 867
empty statement (`;`) 84, 104, 157
`Enabled` property 485
`Enabled` property of a `trace` element 1028
`EnabledViewState` attribute 954
`EnableSession` property of a `WebMethod` attribute 1065
`EnableSessionState` attribute 954
encapsulate 281
encapsulation 402
encoding 1350
`encoding` declaration 1243
encoding scheme 28
encrypt 1020
end of data entry 109
end-of-file marker 759
end of session message 1244, 1247
end tag 840, 1433
`EndElement` constant of enumeration `XmlNodeType` 849
`EndsWith` method of class `String` 641
Enlarged icons in the development window 1219
Enlarging icons using the **Customize** feature 1219
enqueue operation of queue 327, 1165
`EnsureCapacity` method of class `StringBuilder` 652
Enter (or *Return*) key 64, 71
`Enter` method of class `Monitor` 599, 607, 614, 615, 626
entity
  `&amp;` 867

`&lt;` 867
entity reference 867, 1445
entry point 289
entry point of a control structure 166
entry point of a program 63, 66
entry point of control structure 99
entry-level position 1290
`<enumerate>` tag 1242
enumerator 1194
envelope (SOAP) 16
environment variable 1405
equal likelihood 203
Equality and relational operators 81
"equals equals" (the **`==`** operator) 81
`Equals` method of class **`String`** 638
`Equation.cs` 1093
`Error` 148
error handling 74
error-processing code 441
`Error` property of class **`Console`** 759
`ErrorMessage` property 978
  in a Web Form 1020
escape character 918
escape sequence **`\n`** 67, 73
European Computer Manufacturer's Association (ECMA) 18
Event 481
event 23, 185, 479
event argument 481
event driven 479
event-driven process 688
event-driven programming 2
event handler 185, 1247
event handler, create 481
event handler, documentation 481
event handler, register 483
event handler, remove 484
Event-handling model using delegates 479
Examples
  Event-handling model using delegates 479
event handling model 479
Event icon 42
event multicasting 484
events at an interval 580
Events section of the **Properties** window 480
Events window in Visual Studio.NET 480
eWork® Exchange 1300
examination-results problem 118
Examples
  532
  A binary search tree containing 12 values 1169
  A graphical representation of a binary tree 1169
  A graphical representation of the **`InsertAtBack`** operation 1158
  A graphical representation of the **`InsertAtFront`** operation 1158
  A graphical representation of the **`RemoveFromBack`** operation 1160
  A graphical representation of the **`RemoveFromFront`** operation 1159
  A picture with links anchored to an image map 1416
  A portion of a **`Shape`** class hierarchy 347
  Abstract class **`Employee`** definition 404
  Abstract **`CShape`** base class 394
  Abstract **`Shape`** base class 394
  Accessing and displaying a database's data 921
  Accessing **`private`** class members from client code generates syntax errors 291
  ActiveX COM control integration in Visual Basic .NET 1365
  Acts designed to ensure Internet access for people with disabilities 1214
  Add Reference dialog 70
  Adding a new label to the form 48
  Addition program to add two values entered by the user 72
  `Addition.cs` 72
  `AddressBook.cs` 930
  `AdRotator` class demonstrated on a Web form 972
  `AdRotator.aspx` 972
  `AdRotator.aspx.cs` 973
  `AdRotatorInformation.xml` 974
Examples
  `AdvertisementFile` used in **`AdRotator`** example 974
  `Agent.cs` 742
  Airline Reservation Web service 1076
  An illustration of font metrics 699
  `Analysis.cs` 118
  Anchoring demonstration 486
  Animation of a series of images 720
  `Append` methods of class **`StringBuilder`** 655
  Application that writes an XML representation of a **`DataSet`** to a file 939
  Application with accessibility features 1226
  Arc-method demonstration 705
  `ArrayListTest.cs` 1189
  `ArrayReferenceTest.vb` 254
  ASCX code for the header 1015
  ASPX file for the guest book application 1007
  ASPX file that allows a user to select an author from a drop-down list 1021
  ASPX file that presents a list of programming languages 989
  ASPX file that takes ticket information 1078
  ASPX listing for the guest book page 1007
  ASPX page that displays book information 994
  ASPX page that displays the Web server's time 953
  ASPX page with tracing turned off 1028
  Assembly **`TimeLibrary`** used from class **`AssemblyTest`** 333
  `Authors` table of **`books`** database 898
  `Authors.aspx` 1021
  `Authors.aspx.cs` 1023
  `Average1.cs` 107
  `Average2.cs` 112
  `BankUI.cs` 772
  Base class for GUIs in our file-processing applications 772

Examples
- Binary search of a sorted array 262
- Binary search tree 1176
- **BinaryTreeLibrary.cs** 1170, 1178
- **BitArrayTest.cs** 1508
- **BitOperations.cs** 1501
- **BitShift.cs** 1505
- BizTalk markup using an offer Schema 885
- BizTalk terminologies 885
- Blackjack game that uses **Blackjack** Web service 1067
- **Blackjack** Web service 1063
- **Blackjack.cs** 1067
- **BlackjackService.asmx.cs** 1063
- **book.xsd** 872
- **bookxdrfail.xml** 876
- **bookxsd.xml** 872
- **bookxsdfail.xml** 876
- **Boss** class inherits from class **Employee** 405
- **break** statement in a **for** structure 158
- **BreakTest.cs** 158
- Bubble sort using delegates 426
- Bubble-sort **Form** application 427
- **BubbleSortForm.cs** 427
- Business letter DTD 866
- **Button** properties and events 490
- Buttons for message dialogs 148
- C# program for examination-results problem 118
- C#'s view of an n-byte file 760
- Calculating compound interest with **for** 149
- Calculating factorials with a recursive method 217
- CallXML elements 1248
- CallXML example that reads three ISBN values 1245
- **Card** class 664
- Card dealing and shuffling simulation 665
- **Card.cs** 664

Examples
- **CCircle5** class inherits from class **CPoint3** and overrides a finalizer method 374
- Centering the text in the label 50
- Changing a property in the code view editor editor 128
- Changing a property value at run time 130
- **Char**'s **static** character-testing methods and case-conversion methods 661
- Character classes 669
- **CharMethods.cs** 661
- **CheckBox** properties and events 498
- **CheckBoxTest.cs** 499
- **CheckedListBox** and **ListBox** used in a program to display a user selection 541
- **CheckedListBox** properties, methods and events 540
- **CheckedListBoxTest.cs** 541
- Chess-game code 725
- **ChessGame.cs** 725
- **ChessPiece.cs** 723
- **Child** class for MDI demonstration 574
- **Child.cs** 574
- **Circle** class contains an *x-y* coordinate and a radius 351
- **Circle** class marked up with XML comments 1333
- **Circle** class that inherits from class **Point** 385
- **Circle2** class that inherits from class **Point** 355
- **Circle2** class that inherits from class **Point2** 397
- **Circle3** class that inherits from class **Point2** 359
- **Circle3** class that inherits from class **Point3** 421
- **Circle3.cs 421**
- **Circle4** class that inherits from class **Point3** but does not use **protected** data 364
- **CircleTest** class marked up with XML comments 1336

Examples
- **CircleTest** demonstrates class **Circle** functionality 354
- **CircleTest3** demonstrates class **Circle3** functionality 360
- **CircleTest4** demonstrates class **Circle4** functionality 366
- **CircularBuffer.cs** 617
- **CityWeather.cs** 1085
- Class average program with counter-controlled repetition 107
- Class **Control** layout properties 487
- Class **Control** properties and methods 485
- Class **FrmInheritance**, which inherits from class **Form**, contains a button (**Learn More**) 575
- Class **FrmVisualTest**, which inherits from class **VisualForm.FrmInheritance**, contains an additional button 577
- Class that stores equation information 1093
- Class that stores weather information about a city 1085
- Class-average program with sentinel-controlled repetition 112
- Classes that derive from class **Brush** 690
- Client portion of a client/server stream-socket connection 1114
- **Client.cs** 1087, 1114, 1132
- **ClockUserControl.cs** 580
- Code generated by the IDE for **lblWelcome** 127
- Code-behind file for a page that displays the Web server's time 955
- Code-behind file for page demonstrating the **AdRotator** class 973
- Code-behind file for the guest book application 1009

Examples
  Code-behind file for the log in page for authors application 1016
  Code-behind file for the page that allows a user to choose an author 1023
  Code-behind file for the reservation page 1079
  Code-behind file for the word generator page 979
  Code-behind file that writes cookies to the client 991
  `Color` 689
  `Color` structure members 689
  `ColorDialog` used to change background and text color 693
  COM DLL component in Visual Basic.NET 1369
  `ComboBox` demonstration 543
  `ComboBox` properties and events 543
  `ComboBox` used to draw a selected shape 544
  `ComboBoxTest.cs` 544
  `CommissionWorker` class inherits from class **`Employee`** 406
  Common Form properties and events 478
  Complex XHTML table 1460
  `ComplexNumber.cs` 430
  Components and controls for Windows Forms 477
  Computing the sum of the elements of an array 242
  `Concat static` method 648
  Console windows with tabs and without tabs 1224
  `contact.html` 1381, 1386, 1440, 1445
  Container class for chess pieces 723
  `continue` statement in a **`for`** structure 21, 158, 159
  `ContinueTest.cs` 159
  Cookies being read from a client in an ASP .NET application 995
  Cookies getting created in an ASP .NET Web application 991

Examples
  Cookies recieved in an ASP .NET Web application 995
  Counter-controlled repetition with the **`for`** structure 142
  Counter-controlled repetition with **`while`** structure 141
  Create and write to a sequential-access file 777
  `CreateRandomAccessFile.cs` 799
  `CreateSequentialAccessFile.cs` 777
  Creating a new Windows application 45
  Creating a **`Panel`** with scrollbars 496
  Creating files for random-access file-processing applications 799
  Credit-inquiry program 788
  `CreditInquiry.cs` 788
  Custom control added to a **`Form`** 583
  Custom control added to the **ToolBox** 582
  Custom control creation 579
  `Cylinder` class inherits from class **`Circle4`** and overrides method Area 368
  `Cylinder` class inherits from class **`Circle4`** and overrides method **`Area`** 368
  `Cylinder2` class inherits from class **`Circle2`** 398
  `Cylinder3` class inherits class **`Circle3`** 423
  `Cylinder3.cs` 423
  Data hierarchy 759
  Database information input into a DataGrid 1023
  `Date` class encapsulates day, month and year information 306
  Debug sample program 1313
  `DebugClass.cs` 1325
  `DebugExample.cs` 1313
  Debugging methods 1322
  `DeckOfCards.cs` 665
  Default namespaces demonstration 846
  `DeleteDialog.cs` 830

Examples
  `DeleteDialogForm` class enables users to remove records from files in transaction-processor case study 830
  Demonstrating keyboard events 513
  Demonstrating order in which constructors and destructors are called 376
  Demonstrating **`ref`** and **`out`** parameters 198, 452
  Demonstrating that **`finally`** blocks always execute 452
  Demonstrating the bitwise AND, bitwise inclusive OR, bitwise exclusive OR and bitwise complement operators 1501
  Demonstrating the linked list 1155
  Demonstrating the logical operators 164
  Demonstrating window auto-hide 41
  Demonstrating XML namespaces 844
  Demonstration of methods that draw lines, rectangles and ellipses 702
  Details of **`Click`** event 483
  `Directory` class methods (partial list) 762
  Displaying multiple lines in a dialog 68
  Displaying the bit representation of an integer 1498
  `DisplayLogoForm.cs` 718
  `DisplayQueryResults.cs` 928
  `DivideByZeroTest.cs` 443
  `do/while` repetition structure 156
  docking demonstration 487
  DOM structure of an XML document illustrated by a class 851
  `DoWhileLoop.cs` 156
  `DrawArcs.cs` 705
  `DrawPolygons.cs` 708
  `DrawShapes.cs` 712
  `DrawStarsForm.cs` 715

Examples
- DTD for a business letter 866
- **Dynamic Help** window 44
- Ellipse bounded by a rectangle 704
- `Employee` class encapsulates employee name, birthday and hire date 308
- `EmployeesTest` class tests the `Employee` class hierarchy 412
- Enlarged icons in the development window 1219
- Enlarging icons using the **Customize** feature 1219
- Equality and relational operators 81
- `Equation.cs` 1093
- Events section of the **Properties** window 480
- Exception handlers for `FormatException` and `DivideByZeroException` 443
- `Exception` properties and demonstrating stack unwinding 458
- Execute SQL statements on a database 928
- Execution of the `Welcome1` program 66
- Expanded and checked menus 522
- `File` class methods (partial list) 761
- `FileSearch.cs` 766
- `FileTest.cs` 762
- First program in C# 60
- Flowcharting a typical `for` repetition structure 146
- Flowcharting the `do`/`while` repetition structure 157
- `Font` class read-only properties 696
- **Font** window 49
- `FontFamily` class used to obtain font-metric information 699
- `FontFamily` methods that return font-metric information 699
- `Fonts` and s 697
- `ForCounter.cs` 142
- `ForEach.vb` 272
- Form including radio buttons and drop-down lists 1469

Examples
- Form including textareas, password boxes and checkboxes 1406
- Form with sizing handles 47
- `form.html` 1463
- `form2.html` 1466
- `form3.html` 1469
- Framed Web site with a nested frameset 1423, 1484
- GDI+ coordinate system. Units are measured in pixels 687
- `Generator.asmx.cs` 1095
- `Generator.aspx` 977
- `Generator.aspx.cs` 979
- `Generator.html` 984
- `GetHashCode` method demonstration 642
- Graphical representation of a linked list 1149
- `Graphics` methods for drawing arcs 705
- `Graphics` methods for drawing polygons 708
- `Graphics` methods that draw lines, rectangles and ovals 702
- `GroupBox` properties 495
- `GroupBoxPanelExample.cs` 496
- Handling a divide-by-zero exception 443
- `HashtableTest.cs` 1200
- Header elements `h1` through `h6` 1379, 1437
- `header.html` 1379, 1437
- `hello.xml` 1243
- Hierarchical boss method/worker method relationship 180
- Home page written in VoiceXML 1236
- `HourlyWorker` class inherits from class `Employee` 410
- HTML and ECMAScript sent to the client browser 984
- `HugeInteger` Web service 1046
- `HugeInteger.asmx.cs` 1046
- `ibiztalkmarkup.xml` 885
- Icons for message dialogs 148

Examples
- IDE in run mode, with the running application in the foreground 52
- IDE showing program code for a simple program 125
- **Image Collection Editor** window for an `ImageList` component 554
- Image resizing 718
- Image with links anchored to an image map 1476
- `ImageHeader.ascx` 1015
- Important methods of class `HttpCookie` 997
- `index.html` 1480
- `index2.html` 1484
- Indexers provide subscripted access to an object's members 320
- Inheritance examples 345, 346
- Inheritance hierarchy for university `CommunityMembers` 346
- Initializing element arrays in three different ways 240
- Initializing multidimensional arrays 267
- Inserting and aligning the picture box 50, 51
- Inserting special characters into HTML 1386
- Inserting special characters into XHTML 1445
- `Interfaces2Test.cs` 424
- `InterfacesTest.cs` 417
- `isbn.xml` 1245
- `IShape` interface provides methods `Area` and `Volume` and property `Name` 419
- Keyboard events, delegates and event arguments 512
- `KeyDemo.cs` 513
- Label in position with its `Text` property set 48
- `Label` properties 489
- `LabelTextBoxButtonTest.cs` 490
- `LayoutMdi` enumeration values 570
- `letter.xml` 842
- `letter2.xml` 867
- Linear search of an array 260
- `LinesRectanglesOvals.cs` 702

Examples
  - `LinkedListLibrary.cs` 1151
  - Linking to an e-mail address 1440
  - Linking to an email address 1381
  - Linking to other Web pages 1380, 1438
  - `LinkLabel` properties and events 531
  - `LinkLabels` used to link to a folder, a Web page and an application 532
  - `LinkLabelTest.cs` 532
  - `links.html` 1380, 1388, 1438, 1473
  - List of `Form` events 482
  - `list.html` 1389, 1448
  - `ListBox` and `CheckedListBox` on a form 535
  - `ListBox` on an ASPX page 994
  - `ListBox` properties, methods and events 535
  - `ListBox` used in a program to add, remove and clear items 537
  - `ListBoxTest.cs` 537
  - Listing for `namespace.xml` 844
  - `ListTest.cs` 1155
  - `ListView` displaying files and folders 555
  - `ListView` properties and events 553
  - `ListViewTest.cs` 555
  - Log in Web Form 1013
  - `Login.aspx` 1013
  - `Login.aspx.cs` 1016
  - `LogoAnimator.cs` 720
  - `main.html` 1377, 1432, 1478
  - `main.vxml` 1236
  - `MainMenu` and `MenuItem` properties and events 524
  - Manipulating the `Anchor` property of a control 487
  - Math tutor application 1098
  - MDI parent and MDI child events and properties 567
  - MDI parent window and MDI child windows 566
  - `MediaPlayerTest.cs` 733

Examples
  - Memory location diagram 75
  - `MenuItem` property `MdiList` example 569
  - Menus used for changing text font and color 525
  - `MenuTest.cs` 525
  - `MethodDebugExample.cs` 1322
  - Microsoft Agent demonstration 742
  - Microsoft Schema file that contains structure to which `bookxdr.xml` conforms. 871
  - Minimized and maximized child windows 568
  - Miscellaneous `String` methods `Replace`, `ToLower`, `ToUpper`, `Trim` and `ToString` 649
  - Modifying a database 930
  - Mouse events, delegates and event arguments 509
  - `nav.html` 1384, 1443, 1482
  - `NegativeNumberException.cs` 463
  - Nested and ordered lists in HTML 1389
  - Nested and ordered lists in XHTML 1448
  - **New Project** dialog 36
  - New `Text` property value reflected in design mode 129
  - `NewDialog.cs` 827
  - `NewDialogForm` class enables users to create records in transaction-processor case study 827
  - News article formatted with XML 839
  - Object debugging example 1325
  - Operators `checked` and `unchecked`, and handling arithmetic overflow 467
  - `OptionPage.aspx.cs` 991
  - Options supplied on an ASPX page 989, 997
  - `OptionsPage.aspx` 989, 997
  - `OptionsPage.aspx.cs` 999
  - `Overflow.cs` 467

Examples
  - Overloaded constructor demonstration 295
  - Overloaded constructors provide flexible object-initialization options 293
  - `Painter.cs` 510
  - `Panel` properties 495
  - Passing an array reference using `ByVal` and `ByRef` with an array 254
  - Passing arrays and individual array elements to methods 251
  - `PathNavigator.cs` 858
  - Paths used to draw stars on a form 715
  - Peedy calculating the total 742
  - Peedy flying animation 739
  - Peedy introducing himself when the window opens 737
  - Peedy recounting the order 741
  - Peedy repeating the user's request for anchovies as an additional topping 741
  - Peedy repeating the user's request for Seattle-style pizza 740
  - Peedy waiting for speech input 740
  - Peedy's reaction when he is clicked 738
  - `Person` class implements `IAge` interface 415
  - Picture with links anchored to an image map 1416
  - `picture.html` 1382, 1441, 1476
  - `PictureBox` properties and events 507
  - `PictureBoxTest.cs` 507
  - `PieceWorker` class inherits from class `Employee` 408
  - Placing images in HTML files 1382
  - Placing images in XHTML files 1441
  - `Point` class represents an *x-y* coordinate pair 348, 384
  - `Point` marked up with XML comments 1331
  - `Point2` class inherits from abstract class `Shape` 395

- Examples
  - **`Point2`** class represents an *x*-*y* coordinate pair as **`protected`** data 357
  - **`Point2`** class represents an *x*-*y* coordinate pair as **`protected`** data 357
  - **`Point3`** class implements interface **`IShape`** 420
  - **`Point3`** class uses properties to manipulate its **`private`** data 362
  - **`Point3.cs`** 420
  - **`Point4`** base class contains constructors and finalizer 372
  - **`PointTest`** class demonstrates class **`Point`** functionality 350
  - Polygon-drawing demonstration 708
  - Positive and negative arc angles 705
  - Precedence and associativity chart 166
  - Precedence of arithmetic operators 78
  - **`PrintBits.cs`** 1498
  - Printing on multiple lines with a single statement 67
  - Printing on one line with separate statements 67
  - Producer and consumer threads accessing a circular buffer 617
  - Producer and consumer threads accessing a shared object with synchronization 607
  - Producer and consumer threads accessing a shared object with syncronization 607
  - Producer and consumer threads accessing a shared object without synchronization 602
  - Producer and consumer threads accessing a shared object without syncronization 602
  - Program that demonstrates class **`Array`** 1186
  - Program that prints histograms 243
  - Program to display hidden text in a password box 490
- Examples
  - Program to simulate the game of craps 208
  - Programmer-defined **`Maximum`** method 188
  - Project properties dialog 582
  - Properties demonstration for class **`Time3`** 301
  - Properties of class **Control** related to accessibility 1225
  - **Properties** window 43, 49
  - **`Properties.cs`** 458
  - Publication page of Deitel and Associates' VoiceXML page 1238
  - Publication page of Deitel's VoiceXML page 1238
  - **`publications.vxml`** 1238
  - Quantifiers used regular expressions 672
  - **`QueueInheritanceLibrary.cs`** 1166
  - **`QueueTest.cs`** 1167
  - **`RadioButton`** properties and events 502
  - **`RadioButtonsTest.cs`** 502
  - Random-access file with fixed-length records 795
  - **`RandomAccessRecord.cs`** 795
  - Reading records from random-access files sequentially 807
  - Reading sequential-access files 784
  - **`ReadRandomAccessFile.cs`** 807
  - **`ReadSequentialAccessFile.cs`** 784
  - Receiving temperature and weather data from a Web service 1087
  - **`RecommendationsPage.aspx`** 994, 1003
  - RecommendationsPage.aspx.cs 1004
  - **`RecommendationsPage.aspx.cs`** 995
  - Record for random-access file-processing applications 795
  - Record for sequential-access file-processing applications 774
  - **`Record.cs`** 774
- Examples
  - Record-transaction class for the transaction-processor case study 812
  - Recursive evaluation of 5! 217
  - Recursively generating Fibonacci numbers 219
  - **`Regex`** methods **`Replace`** and **`Split`** 677
  - **`Regex`** methods **`Replace and Split`** 677
  - **`RegexMatches.cs`** 669
  - **`RegexSubstitution.cs`** 677
  - Regular expression used to determine file types 766
  - Regular expressions checking birthdays 669
  - Relational-database structure of an **`Employee`** table 897
  - Removing tabs from Visual Studio environment 1223
  - **`Reservation.asmx.cs`** 1076
  - Rolling dice in a windows application 203
  - Sample data for the program of Fig. 17.9 783
  - Sample Internet Explorer window with GUI components 476
  - Saving a document to a file 1340
  - Schema-validation example 874
  - Scoping example 213
  - SDI and MDI forms 566
  - Searching for characters and substrings in **`String`**s 643
  - Selecting the **Build Comment Web Pages** from **Tools** menu 1339
  - Self-referential class objects linked together 1148
  - Server portion of a client/server stream-socket connection 1111
  - Server side of client/server Tic-Tac-Toe program 1126
  - **`Server.cs`** 1111, 1120
  - Server-side portion of connectionless client/server computing 1120

Examples
- Session data read by an ASP .NET Web application to provide recommendations for the user 1004
- Session information displayed in a **ListBox** 1003
- Sessions are created for each user in an ASP .NET Web application 999
- Sessions created for each user in an ASP .NET Web application 999
- Set of recursive calls to method **Fibonacci** 221
- Setting the form's **Text** property 46
- Setting the project location 46
- Shapes drawn on a form 712
- Shifted random integers 201
- ShowColors.cs 690
- Sieve of Eratosthenes 1508
- Simple Class Library 332
- Simple form with hidden fields and a text box 1463
- Simple program as it executes 45
- Simple student-poll analysis program 248
- **SimpleEventExample.cs** 481
- Simulating rolling 12 six-sided dice 205
- SOAP Request for the **HugeInteger** Web service 1045
- **Solution Explorer** window 40
- Some basic GUI components 476
- Some common escape sequences 68
- Some methods of class **ArrayList** 1189
- Sorting an array with bubble sort 257
- **sorting.xsl** 879
- **sports.xml** 864
- SQL query keywords 905
- **SquareRootTest.cs** 464
- **StackComposition** class encapsulates functionality of class **List** 1164
- **StackComposition-Library.cs** 1164

Examples
- **StackInheritance** extends class **Lis** 1161
- **StackInheritance-Library.cs** 1161
- **StackInheritance-Test.cs** 1162
- **StackTest.cs** 1195
- **Start Page** in Visual Studio .NET 35
- **StartDialog.cs** 817
- **StartDialogForm** class enables users to access dialog boxes associated with various transactions 817
- **StartsWith** and **EndsWith** methods 641
- **static** member demonstration 315
- **static** members are accessible to all objects of a clas 314
- **Static** method **Concat** 648
- **String** constructors 635
- **string** formatting codes 151
- **String** indexer, **Length** properties and **CopyTo** method 636
- **String** methods **Replace**, **ToLower**, **ToUpper** and **Trim** 649
- **String** methods **Replace**, **ToLower**, **ToUpper**, **Trim** and **ToString** 649
- **String** test to determine equality 639
- **StringBuilder** class constructors 651
- **StringBuilder** size manipulation 653
- **StringBuilder** text insertion and removal 658
- **StringBuilder** text replacement 659
- **StringBuilder**'s **AppendFormat** method 656
- **StringBuilder-Append.cs** 655
- **StringBuilderAppend-Format.cs** 656
- **StringBuilder-Constructor.cs** 651
- **StringBuilder-Features.cs** 653

Examples
- **StringBuilderInsert-Remove.cs** 658
- **StringBuilder-Replace.cs** 659
- **StringCompare.cs** 639
- **StringConstructor.cs** 635
- **StringHashCode.cs** 642
- **StringIndexMethods.cs** 643
- **StringMethods.cs** 636
- **StringMiscellaneous2.cs** 649
- **StringStartEnd.cs** 641
- **SubConcatination.cs** 648
- **SubString.cs** 646
- Substrings generated from **String**s 646
- **Sum.cs** 147
- Summation using **for** 147
- **SwitchTest.cs** 152
- **Synchronized.cs** 607
- Syntax error generated from overloaded methods 225
- **System.Drawing** namespace's classes and structures 686
- Tabbed pages in Visual Studio .NET 560
- **TabControl** properties and events 561
- **TabControl** used to display various font settings 562
- **TabControl** with **TabPages** example 561
- Table optimized for screen reading using attribute **headers** 1232
- **table1.html** 1457
- **table2.html** 1460
- **TableDisplay.cs** 921
- **TabPages** added to a **TabControl** 561
- **TemperatureServer** Web service 1082
- **TemperatureServer.asmx.cs** 1082
- **Test2** demonstrates polymorphism in Point-Circle-Cylinder hierarchy 400

Examples
- **`Test3`** uses interfaces to demonstrate polymorphism in Point-Circle-Cylinder hierarchy 424
- Testing class **`Cylinder`** 370
- **`TextBox`** properties and events 489
- **`TextToSpeech.cs`** 1226
- **`this`** reference demonstration 311
- **`this`** reference used implicitly and explicitly to enable an object to manipulate its own data and invoke its own 310
- Thread life cycle 593
- thread-priority scheduling 596
- Threads sleeping and printing 596
- **`ThreadTester.cs`** 596
- **`TicketReservation.aspx`** 1078
- **`TicketReservation.aspx.cs`** 1079
- **`Time1`** abstract data type represents the time in 24-hour format 283
- **`Time1.cs`** 283
- Tool tip demonstration 39
- **Toolbox** window 41
- **`Transaction.cs`** 812
- **`TransactionProcessor.cs`** 816
- **`TransactionProcessorForm`** class runs the transaction-processor application 816
- **`TransformTest.cs`** 882
- Tree structure for Fig. 18.1 847
- **`Tree.cs`** 416
- **TreeNode Editor** 550
- **`TreeNode`** properties and methods 549
- **`TreeTest.java`** 1174
- **`TreeTestLibrary.cs`** 1182
- **`TreeView`** displaying a sample tree 548
- **`TreeView`** properties and events 548
- **`TreeView`** used to display directories 550
- **`TreeViewDirectoryStructureTest.cs`** 550

Examples
- Truth table 161
- **`Tutor.cs`** 1098
- Unordered lists in HTML 1388
- Unordered lists in XHTML 1447
- **`Unsynchronized.cs`** 602
- **`UpdateDialog.cs`** 822
- User-defined exception classes 464
- Using **`<meta>`** and **`<DOCTYPE>`** 1418
- Using a **`PictureBox`** to display images 507
- Using an abstract data type 287
- Using arrays to eliminate a **`switch`** structure 245
- Using **`CheckBox`**es to change font styles 499
- Using class **`StackInheritance`** 1162
- Using default namespaces 846
- Using **`For Each`**/**`Next with an array`** 272
- Using **`GroupBox`**es and **`Panel`**s to arrange **`Button`**s 496
- Using images as link anchors 1384, 1443
- Using inheritance to create a queue 1167
- Using internal hyperlinks to make your pages more navigable 1413, 1473
- Using **`meta`** to provide keywords and a description 1478
- Using operator overloading 433
- Using overloaded methods 223
- Using **`RadioButton`**s to set message-window options 502
- Using the **`HugeInteger`** Web service 1057
- Using the mouse to draw on a form 510
- Using the **`Properties`** window to set a property value 127
- Using the **`Stack`** class 1195
- **`UsingArray.cs`** 1186
- **`UsingExceptions.cs`** 452

Examples
- **`UsingFontMetrics.cs`** 699
- **`UsingFonts.cs`** 697
- **`UsingHugeIntegerService.cs`** 1057
- **`UsingMDI.cs`** 571
- **`UsingTabs.cs`** 562
- **`Validate.cs`** 672
- Validating user information using regular expressions 672
- **`ValidationTest.cs`** 874
- Validators used in a Web Form that generates possible letter combinations from a phone number 977
- Viewing the tracing information for a project 1029
- Visual Inheritance through the Form Designer 577
- Visual Studio .NET environment after a new project has been created 37
- Visual Studio .NET menu bar 38
- Visual Studio .NET **Menu Designer** 523
- Visual Studio .NET menu summary 38
- **`VisualInheritance.cs`** 575
- **`VisualInheritanceTest.cs`** 577
- VoiceXML tags 1242
- Web controls demonstration 967
- Web service that generates random equations 1095
- Web site using two frames: navigational and content 1420, 1480
- **`WebControls.aspx`** 967
- **`Welcome.aspx`** 1007
- **`Welcome.aspx.cs`** 1009
- **`Welcome1.cs`** 60
- **`Welcome2.cs`** 67
- **`Welcome3.cs`** 67
- **`Welcome4.cs`** 68
- **`WhileCounter.cs`** 141
- Windows Form Designer generated code expanded 126
- Windows Form Designer generated code reflecting new property values 128

Examples
  Windows Media Player demonstration 733
  withheaders.html 1232
  withoutheaders.html 1231
  WriteRandomAccessFile.cs 802
  Writing records to random-access files 802
  XHTML document displayed in the left frame of Fig. 5.9. 1482
  XHTML documentation of class **Circle** 1340
  XHTML documentation of method **Area** method of class **Circle** 1341
  XHTML table 1457
  XHTML table without accessibility modifications 1231
  XML document containing book information 878
  XML document referencing its associated DTD 867
  XML document that conforms to W3C XML Schema. 872
  XML document that describes various sports 864
  XML document that does not conform to XSD schema 876
  XML documentation generated by Visual Studio .NET 1341
  XML file containing **AdRotator** information 974
  XML file that does not conform to the Schema in Fig. 18.17 876
  XML namespaces demonstration 844
  XML to mark up a business letter 842
  XML used to mark up an article 839
  XML Validator displaying an error message 870
  XML Validator used to validate an XML document 869
  XmlDom.cs 851
  XmlNodeReader used to iterate through an XML document 848
  XmlReaderTest.cs 848

Examples
  XMLWriter.cs 939
  XPath expressions and descriptions 865
  XPathNavigator class used to navigate selected nodes 858
  XSD Schema document to which **bookxsd.xml** conforms 872
  XSL document that transforms **sorting.xml** into XHTML. 879
  XSL style sheet applied to an XML document 882
exception 23, 249, 439
Exception class 442, 448, 451, 457
exception handler 439, 442, 446, 448
Exception handlers for **FormatException** and **DivideByZeroException** 443
Exception library class 23
Exception properties and demonstrating stack unwinding 458
exception thrown within a SyncBlock 626
Exclamation 148
.exe extension 69
Execute SQL statements on a database 928
ExecuteNonQuery method of **OleDbCommand** 938
Execution of the **Welcome1** program 66
execution stack 1160
exhausting memory 219
Exists method of class **Directory** 762
Exit method of class **Application** 529, 539
Exit method of class **Environment** 1119
Exit method of class **Monitor** 599, 607, 626
Exit method of **Monitor** 615, 616
exit point of a control structure 166
exit point of control structure 99
<exit> tag 1242
Exp method of class **Math** 182
Expand method of class **TreeNode** 549

expand node 547
expand tree 39
ExpandAll method of class **TreeNode** 549
Expanded and checked menus 522
expanded code 126
Experience.com 1302
Expires property of a cookie 988
Expires property of **HttpCookie** class 997
explicit conversion 115, 193
explicit relationships between data 841
exponential “explosion” of calls 222
exponential method 182
exponentiation 77, 182
exponentiation operator 150
expose a method 1052
exposing a Web service method 1041
expression 1316
Extended Backus-Naur Form (EBNF) grammar 866, 870
extensibility 15
Extensible HyperText Markup Language (XHTML) 15, 25, 28, 1431
extensible language 287
Extensible Linking Language (XLink) 16
Extensible Markup Language (XML) 15, 1042
Extensible Stylesheet Language (XSL) 16, 844, 877
Extensible Stylesheet Language Transformation (XSLT) 25
external DTD 868
external help 43
Extra Keyboard Help 1257

F

F formatting code 151
F1 help key 43
factorial 176, 216
fall through 154
false 100
falsity 80
fatal logic error 104
fault-tolerant program 439
favorites 1339
FCL (Framework Class Library) 12, 18, 20, 27, 179, 327
Fibonacci series 219, 221

field 282, 758, 897, 898
FIFO (first-in, first-out) 327
file 758
file as a collection of bytes 758 761
`File` class methods (partial list) 761
File menu 38, 69
file-position 788
file-position pointer 788
file processing 757
file synchronization 17
`FileAccess` enumeration 782
`FileName` property of class `AxMediaPlayer` 735
file-processing programs 760
files 757
`FileSearch.cs` 766
`FileStream` class 760, 781, 787
`FileTest.cs` 762
`Fill` method of class `Graphics` 717
`Fill` method of `OleDbAdapter` 928
`<filled>` tag 1242
`FillEllipse` method of class `Graphics` 547, 702
`FillPie` method of class `Graphics` 547, 705
`FillPolygon` method of class `Graphics` 708
`FillRectange` method of class `Graphics` 547, 693, 701, 702
final value of control variable 140, 141, 143, 146
`Finalize` method 372
finalizer 312
`finally` block 442, 450
find 38
firewall 1045
First program in C# 60
first refinement 116
first-in, first-out (FIFO) data structure 327, 1165
`FirstNode` property of class `TreeNode` 549
five-pointed star 715
fixed-length records 794
flag value 109
`FlipDog.com` 1291
`float` primitive data type 73, 196
`float` promotions 194
floating-point division 115
floating-point number 113
`Floor` method of class `Math` 182
flow of control 85, 105, 114
flowchart 21, 97, 100
flowchart of `for` statement 145
flowchart reducible to the simplest flowchart 170
flowchart symbol 166
Flowcharting a double-selection `if`/`else` structure 101
Flowcharting a single-selection `if` structure 100
Flowcharting a typical `for` repetition structure 146
Flowcharting C#'s sequence structure 97
Flowcharting the `do`/`while` repetition structure 157
Flowcharting the `switch` multiple-selection structure 155
Flowcharting the `while` repetition structure 106
`FlowLayout` 963
flowline 97, 100
`Focus` method 477, 485
`Focused` property 485
font 686, 696
`Font` class 686, 697
`Font` class read-only properties 696
font control 696
font descent 698
font height 698
font leading 698
font manipulation 687
font metrics 698
font name 697
`Font` property 49, 485, 500
font size 49, 697
font style 49, 499, 697
Font window 49
`FontFamily` class 686, 699
`FontFamily` class used to obtain font-metric information 699
`FontFamily` methods that return font-metric information 699
`FontFamily` property of class `Font` 696
`Font`s and `FontStyle`s 697
`FontStyle` enumeration 697
`for` flowchart 146
`for` repetition structure 21, 98, 142, 143, 144, 145, 146, 150, 167, 171
`for` structure header 143, 144, 149
`ForCounter.cs` 142
`foreach` repitition structure 98, 171
`ForEach.vb` 272
`ForeColor` property 485
foreign key 904
form 37, 476, 1398, 1403, 1457, 1462
`Form` array 983
form background color 47
`Form` class 124, 478
`Form Close` method 478
`form` element 1241, 1242, 1405, 1463
Form including textareas, password boxes and checkboxes 1406
form input 1413
`Form` properties and events 478
`Form` property `IsMdiContainer` 566
form title bar 46
Form with sizing handles 47
format 75
`format` attribute 1249
Format menu 38
`Format` method of `String` 286
format specifications 286
format string 286, 657
`FormatException` class 445, 446
formatting code 151
forming structured programs 166
`forms` 1021
forms authentication 1020
`FormsAuthentication` class 1020
FORmula TRANslator (Fortran) 10
Fortran (FORmula TRANslator) progamming language 10, 19, 1431
forward slash (`/`) 840, 1443
`ForwardDiagonal` member of enumeration `LinearGradientMode` 714
fractional result 114
frame 1234, 1420, 1479
`frame` element 1422, 1482
Framed Web site with a nested frameset 1423, 1484
frameset document type 1479
`frameset` element 1421, 1422

Framework Class Library (FCL) 12, 18, 20, 27, 179, 327
FreeBSD operating system 18
Freedom Scientific 1248
FROM 905, 909, 910, 911, 913
FromArgb method of structure **Color** 689
FromImage method of class **Graphics** 714
FromName method 689
FullName property 559
FullPath property of class **TreeNode** 549
fully qualified name 913
function 62
functionalization 3
Futurestep.com 1297

G

G formatting code 151
G property of structure **Color** 690
gallery.yahoo.com 1441
game-playing 200
game-playing program 200
garbage collection 312, 592
garbage collector 312, 372, 636
garbage-collector thread 592
Gates, Bill 8
GC class 316
GDI+ (Graphics Device Interface+) 24, 685
687
general path 715
Generator.asmx.cs 1095
Generator.aspx 977
Generator.aspx.cs 979
Generator.html 984
Genie **Microsoft Agent** character 736
get accessor 291, 297, 298, 301
Get method of class **BitArray** 1510
get request type 1248, 1464
GetCellAscent method of class **FontFamily** 699
GetCellDescent method of class **FontFamily** 699
GetCreationTime method of class **Directory** 762
GetCreationTime method of class **File** 761
GetCurrentDirectory method 508
getDigits element 1247
GetDirectories method of class **Directory** 552, 762
GetDirectories method of class **DirectoryInfo** 559
GetEmHeight method of class **FontFamily** 699
GetEnumerator method of **ArrayList** 1194
GetEnumerator method of **Hashtable** 1204
GetFiles method of class **Directory** 762
GetFiles method of class **DirectoryInfo** 559
GetHashCode method demonstration 642
GetHashCode method of class **Object** 1200
GetHashCode of class **String** 642
GetItemChecked method of class **CheckedListBox** 540
GetLastAccessTime method of class **Directory** 762
GetLastAccessTime method of class **File** 761
GetLastWriteTime method of class **Directory** 762
GetLastWriteTime method of class **File** 761
GetLineSpacing method of class **FontFamily** 699
GetNodeCount method of class **TreeNode** 549
GetSelected method of class **ListBox** 536
GetStream method of class **Socket** 1110
GetXml method of **DataSet** 938
global scope 213
global variable 313, 1244
Globally Unique Identifier (GUID) 1367
glyph 1352
golden mean 219
golden ratio 219
Good Programming Practice 12
Gosling, James 9
goto elimination 97
goto statement 97, 168, 1248
<goto> tag 1242
<grammar> tag 1242
graph information 243
Graphical representation of a linked list 1149
graphical representation of an algorithm 97
graphical user interface (GUI) 23, 37, 69, 475
Graphics class 544, 685, 687, 697, 701, 714, 715, 717
graphics context 687
Graphics Device Interface+ (GDI+) 24, 685
Graphics Interchange Format (GIF) 1441
Graphics methods for drawing arcs 705
Graphics methods for drawing polygons 708
Graphics methods that draw lines, rectangles and ovals 702
Graphics property of class **PaintEventArgs** 579
GraphicsPath class 715
GraphicsUnit structure 696
Gray static property of structure **Color** 689
“grayed-out” 485
greatest common divisor (GCD) 235
greedy quantifier 671
Green project 9
Green static property of structure **Color** 689
GridLayout 963
GridLayout and **FlowLayout** illustration 965
GROUP BY 905
group of related fields 758
GroupBox 495
GroupBox Controls property 495
GroupBox properties 495
GroupBox properties and events 495
GroupBoxPanelExample.cs 496
guest book 1006
Guest-book application GUI 1007
GUI (Graphical User Interface) 23
GUI component 69, 475, 476
GUI event-handling 298
GUID (Globally Unique Identifier) 1367
Gunning Fog Index 1218, 1266

H

`h1` header element 1379, 1380, 1436
`h6` header element 1379, 1436
HailStorm Web services 18
handle event 480
handling errors 74
hardware 3
“has-a” relationship 344
hash code 642
hash table 642
`Hashtable` class 1091, 1185, 1199
`HashtableTest.cs` 1200
Haskell progamming language 19
`HatchBrush` class 690, 711
`HatchStyle` enumeration 690
head 1146, 1433
`head` element 954, 1378, 1433
head of a queue 1165
head section 1433
header 1379, 1436
header cell 1400, 1459
header element 1436
`header.html` 1379, 1437
`headers` attribute 1232, 1234
`headers` element 27
`Headhunter.net` 1298
`height` attribute 1383, 1441, 1442
`Height` property of class `Font` 696
Hejlsberg, Anders 9
Hello World CallXML example 1243
help filter 42
Help menu 38, 42
help, context-sensitive 43
help, dynamic 42
help, external 43
help, internal 43
helper method 285, 1176
Henter-Joyce 1248, 1265
hex code 1387
hexadecimal (base16) number system 176
hexadecimal value 1446
hidden
  element 987
  field 987
hidden input elements 1405
hide an internal data representation 327
`Hide` method of class `Form` 478, 485
hiding implementation 180, 290, 326
hierarchical boss method/worker method relationship 180
hierarchy 841
hierarchy diagram 345
hierarchy of shapes 390
high-level language 6, 7
high-order bit 1498
`Highest ThreadPriority` enumeration member 594
`Hire.com` 1297
`HireAbility.com` 1299
`Hirediversity.com` 1296
histogram 243
hit count 1320
Hoare, C. A. R. 599
home page 1376
Home Page Reader (HPR) 1218
Home page written in VoiceXML 1236
horizontal coordinate 686
horizontal rule 28, 1447
horizontal tab 68
host 950
hostname 950
hot key 522
`HotDispatch.com` 1299
`HotJobs.com` 1294, 1298
hotspot 1416, 1476
`hotwired.lycos.com/webmonkey/00/50/index2a.html` 1451
`HourlyWorker` class inherits from class `Employee` 410
HPR (Home Page Reader) 1218
`hr` element 1388, 1447
`href` attribute 1381, 1385, 1416, 1439, 1475
`.htm` (html file extension) 1376
`.html` (html file name extension) 1376
`.html` (XHTML file name extension) 1431
HTML (HyperText Markup Language) 14, 15, 25, 26, 28, 949, 950, 1375, 1431
  `form` 988, 1006
  `HTML` element 1015
HTML and ECMAScript sent to the client browser 984
HTML comment 1377
HTML control 966
HTML document 28
`html` element 954, 1433
HTML frame 28
HTML-Kit 1376
HTML list 28
HTML mode of Web Form designer 963
HTML recommendation 1376
HTML source code 1376
HTML table 28
HTML tag 950, 1375
HTTP (HyperText Transfer Protocol) 20, 950, 1248
HTTP being used with firewalls 1045
HTTP GET request 1043
HTTP header 952
HTTP method 951
HTTP POST request 1043
HTTP transaction 951
`http://www.w3.org/2001/XMLSchema` URI 873
`HttpCookie` class 994, 996, 997
  `Domain` property 997
  `Expires` property 997
  `Name` property 996, 997
  `Path` property 997
  `Secure` property 997
  `Value` property 996, 997
`HttpCookie` properties 997
`HttpCookieCollection` 996
`HttpSession` class 1003
  `SessionID` property 1003
`HttpSessionState` class 997, 999, 1002, 1003, 1006
  `Counts` property 1003, 1006
  `IsNewSession` property 1003
  `IsReadOnly` property 1003
  `Keys` property 1003, 1006
  `SessionID` property 1003
  `Timeout` property 1003
`HttpSessionState` properties 1003
`HugeInteger` Web service 1046
`HugeInteger.asmx.cs` 1046
hyperlink 950, 1380, 1385, 1438
HyperText Markup Language (HTML) 14, 25, 26, 949, 950, 1431
HyperText Transfer Protocol (HTTP) 20, 950
HyTime 16

I

I/O completion 595

IAgentCtlCharacter interface 742, 748, 749
IAgentCtlUserInput interface 750
ìbiztalkmarkup.xml 885
IBM (International Business Machines) 5
IBM Corporation 1351
IBM Personal Computer 5
IComparable interface 1177, 1178
IComponent interface 477
icon 38
Icons for message dialogs 148
IDE (integrated development environment) 9, 20, 23, 34
IDE in run mode, with the running application in the foreground 52
IDE showing program code for a simple program 125
IDE's toolbox and **LabelScrollbar** properties 1365
identifier 62
IDictionaryEnumerator interface 1204
IE (Internet Explorer) 35, 475, 532, 841, 1431, 1442
IEEE 754 floating-point 196, 197
IEnumerator interface 1194, 1204
if selection structure 80, 83, 98, 100, 113, 152, 167, 170
<if> tag (**<if>**…**</if>**) 1242
if/**else** selection structure 21, 98, 100, 117, 152, 167, 170
ignoring array element zero 249
IIS (Internet Information Services) 950
IL (Intermediate Language) 18
image anchor 1413
Image Collection Editor 553
Image Collection Editor window for an **ImageList** component 554
image hyperlink 1385, 1444
image map 28, 1416, 1418, 1477
Image property 51, 507
718
ImageHeader.ascx 1015
ImageIndex property of class **ListViewItem** 553
ImageIndex property of class **TreeNode** 549
ImageList class 553
ImageList collection 548
ImageList property of class **TabControl** 561
ImageList property of class **TreeView** 548
images in Web pages 1382, 1441
ImageUrl attribute 974
img element 27, 1217, 1385, 1418, 1441, 1442, 1444
Immediate window 1318
immutable **String** 636
implement an interface 419
implementation 282, 290
implementation-dependent code 290
implementation of a class hidden from its clients 289
implicit conversion 115, 193
implicitly **sealed** method 402
#IMPLIED flag 867
Important methods of class **HttpCookie** 997
Impressions attribute 974
In property of class **Console** 759
Inch member of enumeration **GraphicsUnit** 696
Inclusive Technology 1261
increasing order 257
increment and decrement operators 121
increment expression 144, 159
increment of a **for** structure 145
increment of control variable 141, 143, 146
increment of loop 140
increment operator, **++** 121
indefinite postponement 595
indefinite repetition 109
indentation 99
indentation convention 101
indentation in **if** statements 84
indentation techniques 63, 85
Indenting each level of a nested list in code 1391
Index event of class **CheckedListBox** 541
Index property of class **MenuItem** 524
index.html 1376
indexer 636
indexer for class **Hashtable** 1091
indexer property 319
Indexers provide subscripted access to an object's members 320
IndexOf method of class **Array** 1188
IndexOf method of class **ArrayList** 1189, 1194
IndexOf method of class **String** 643, 646
IndexOfAny method of class **String** 643
IndexOutOfRangeException class 249, 448
indirect base class 345
infinite loop 105, 109, 114, 144, 158, 219, 1125
infinite recursion 219, 222
infinity symbol 904
Information 148
information hiding 282, 326, 402
information parameter 840
information tier 952
Informix 897
inherit from class **Control** 579
inherit from Windows Form control 579
inherit implementation 437
inherit interface 392, 437
inheritance 22, 282, 284, 290, 343, 346, 377, 383, 419
Inheritance examples 345
inheritance hierarchy 345, 393
Inheritance hierarchy for university **CommunityMembers** 346
Inheritance hierarchy for university **CommunityMembers** 346
inheritance with exceptions 448
inheriting interface versus inheriting implementation 437
Inherits attribute of ASP .NET page 954
Init event 957
initial set of classes 282
initial value of control variable 140, 143, 146
initialization at the beginning of each repetition 119
initialization phase 112
initialization section of the **for** structure 149
initialize instance variables 289
initializer list 242, 266
initializers 292

initializing arrays 240
initializing double-subscripted arrays in declarations 267
Initializing element arrays in three different ways 240
Initializing multidimensional arrays 267
inlining code 402
inner block 213
inner **for** structure 243, 259
INNER JOIN 905, 913
InnerException property of **Exception** 457, 461
innermost pair of parentheses 77
innermost set of square brackets 249
inorder traversal of a binary tree 1170
input 37
input data from the keyboard 476
input device 4
input element 1405, 1406, 1465
input unit 4
input/output 760
input/output blocking 595
input/output operation 98
input/output request 594
inputting data from the keyboard 85
INRIA (Institut National de Recherche en Informatique et Automatique) 15
insert an item into a container object 285
INSERT INTO 905, 917
Insert method of class **ArrayList** 1189
Insert Separator option 523
INSERT statement 917
InsertCommand property of **OleDbAdapter** 927
Inserting and aligning the picture box 50
inserting separators in a menu 523
insertion point 1149
instance of a built-in type 282
instance of a user-defined type 282
instance variable 212, 284, 292, 297, 309, 348, 351, 1338
“instant-access” application 794
instantiate (or create) objects 282
Institut National de Recherche en Informatique et Automatique (INRIA) 15
int primitive data type 73, 112, 196
int promotions 194
Int32.Parse method 75, 114
integer division 77, 114, 115
integer mathematics 326
integer quotient 77
integer value 73
integral data type 196, 466
integrated development environment (IDE) 9, 20, 34
intelligent agent 1291
IntelliSense 129
interactions among objects 326
interactive animated character 24, 736
interface 185, 283, 291, 414, 415, 418, 419, 897
interface definition 1338
interface keyword 414
Interfaces2Test.cs 424
InterfacesTest.cs 417
Intermediate Language (IL) 18
internal data representation 327
internal help 43
internal hyperlink 28, 1413, 1416, 1473, 1475
internal member access modifier 347
International Business Machines (IBM) 5
Internet 14, 15
Internet Explorer (IE) 35, 475, 532, 841, 1431, 1442
Internet Explorer window 69
Internet Information Services (IIS) 950
Internet Protocol (IP) 14
Internet Protocol Addresses (IP Address) 1108
Internet Service Provider (ISP) 1405, 1465
Internshipprograms.com 1302
interpreter 7
Interrupt method of class **Thread** 594
Interval property of class **Timer** 580
InterviewSmart™ 1303
intranet 11, 12, 13
Invalidate method of class **Control** 688
InvalidCastException 388, 389
InvalidOperationException 1198
Invoke 1043
invoking a method 180, 181
invoking a method of a Web service from a Web browser 1044
IP (Internet Protocol) address 14, 950, 1108
IPAddress class 1110
IPEndPoint class 1110
“is-a” relationship 344, 388, 391, 419
IsAccessible property of class **Control** 1226, 1230
isbn attribute 881
IsDigit method of class **Char** 661
IsEmptyElement Property of XmlNodeReader 854
IShape interface provides methods **Area** and **Volume** and property **Name** 419
IsLetter method of class **Char** 661
IsLetterOrDigit method of class **Char** 661
IsLower method of class **Char** 663
IsMdiChild property of class **Form** 567
IsMdiContainer property of class **Form** 566, 567
IsNewSession property of **HttpSessionState** class 1003
ISP (Internet Service Provider) 1405, 1465
IsPostBack property of class **Page** 983
IsPunctuation method of class **Char** 663
IsReadOnly property of **HttpSessionState** class 1003
IsSymbol method of class **Char** 663
IsUpper method of class **Char** 663
IsValid property of **ServerValidateEventArgs** class 1020
IsWhiteSpace method of class **Char** 663
Italic member of enumeration **FontStyle** 697
Italic property of class **Font** 696

ItemActivate event of class **ListView** 554
ItemCheck event of class **CheckedListBox** 540
ItemCheckEventArgs event of class **CheckedListBox** 541
Items property of class **ComboBox** 543
Items property of class **ListBox** 535
Items property of class **ListView** 554
ItemSize property of class **TabControl** 561
iteration 222
iteration of a **for** loop 249
iteration of a loop 140, 144, 159
iterative 219
iterative binary search 262
iterator 394, 1194
iterator class 394

J

Jacopini, G. 97, 170
jagged array 265, 266, 268
Java Development Kit (Java SDK 1.3) 1236
Java progamming language 19
JAWS (Job Access with Sound) 1248, 1265
job 4
jobfind.com 1294
Jobs.com 1295
JobsOnline.com 1298
Join method of class **Thread** 594, 627
joining tables 904
Joint Photographic Experts Group (JPEG) 1441
JScript scripting language 19
JSML 1265
JustCJobs.com 1299
JustComputerJobs.com 1299
JustJavaJobs.com 1290, 1299

K

Keio University 15
Kemeny, John 8
key code 514
key data 514
key event 511, 512
key value 260, 514, 1177
key, modifier 511
keyboard 3, 5, 71, 475
Keyboard events, delegates and event arguments 512
KeyDemo.cs 513
KeyDown event 512
KeyEventArgs properties 512
KeyPress event 512
KeyPressEventArgs properties 512
Keys property of **HttpSessionState** class 1003, 1006
KeyUp event 512
keyword 62, 98
Koenig, Andrew 439
Kurtz, Thomas 8

L

label 48, 49, 69, 71, 475, 476, 488
Label class 488
Label in position with its **Text** property set 48
Label properties 489
labels in a **switch** structure 152
LabelTextBoxButtonTest.cs 490
LAN (local area network) 6
language attribute 954
language independence 19
language interoperability 10, 19
LargeImageList property of class **ListView** 553
LastChild property of **XmlNode** 855, 857
last-in, first-out (LIFO) data structure 1160
LastIndexOf method of class **Array** 1188
LastIndexOf method of class **String** 643, 646
LastIndexOfAny method of class **String** 643
last-in-first-out (LIFO) 326
LastNode property of class **TreeNode** 549
Latin World 1297
layout control 38, 484
layout windows 38
LayoutMdi enumeration 570
LayoutMdi enumeration values 570
LayoutMdi method of class **Form** 567, 570
LayoutMdi.ArrangeIcons 570
LayoutMdi.Cascade 570
LayoutMdi.TileHorizontal 570
LayoutMdi.TileVertical 570
lazy quantifier 671
leaf node in a binary search tree 1169, 1175
left brace (**{**) 63, 71
left child 1169
left-shift assignment operator (**<<=**) 1508
left-shift operator (**<<**) 1496, 1497, 1498, 1504
left subtree 1169, 1175, 1210
left-to-right evaluation 79
length of an array 238
Length property of class **BitArray** 1508
Length property of class **String** 636, 637
Length property of class **StringBuilder** 652
letter 758
letter.dtd 866, 866
letter.xml 842
letter2.xml 867
level of nesting 168, 1391
level of refinement 110, 112
level-order binary tree traversal 1177
lexicographical comparison 638
<li> (list item) tag 1389, 1447
lifetime of an identifier 212
LIFO (last-in, first-out) 326
LIKE 908, 909, 911
likelihood 203
line 685
linear collection 1148
linear data structure 1168
linear search 260, 261, 278
Linear search of an array 260
LinearGradientBrush class 690, 711, 714
LinearGradientMode enumeration 714
linearized 1231
LinesRectanglesOvals.cs 702
link 1148, 1169
link element in VoiceXML 1242
link for a self-referential class 1147
link one Web page to another 1413

<link> tag 1242
LinkArea property of class **LinkLabel** 531
LinkBehavior property of class **LinkLabel** 531
LinkButton 1027
LinkClicked event of class **LinkLabel** 530, 532
LinkColor property of class **LinkLabel** 531
linked document 1380
linked list 26, 285, 394, 1146, 1148, 1149
linked list in sorted order 1149
LinkedListLibrary.cs 1151
LinkLabel class 521, 530
LinkLabel properties and events 531
LinkLabel used to link to a folder, a Web page and an application 532
LinkLabels used to link to a folder, a Web page and an application 532
LinkLabelTest.cs 532
Links property of class **LinkLabel** 531
links.html 1380, 1388, 1438
links2.html 1447
LinkVisited property of class **LinkLabel** 531
Linux operating system 5, 6
list 476
List of **Form** events 482
list, editable 544
list.html 1389, 1448
ListBox and **CheckedListBox** on a form 535
ListBox class 521, 534
ListBox on an ASPX page 994
ListBox properties, methods and events 535
ListBox used in a program to add, remove and clear items 537
ListBox Web control 1003
ListBoxTest.cs 537
Listing for **namespace.xml** 844
ListTest.cs 1155
ListView class 553
ListView displaying files and folders 555
ListView properties and events 553
ListViewTest.cs 555
literal **String** objects 634
live-code approach 2
Load event 478
Load method of **XMLDocument** 850
Load method of **XslTransform** 882
local area network (LAN) 6
local dialog 1241
local variable 181, 212
local variable is destroyed 252
local variable of a method 290
local variables of a method 213, 309
localhost 1119
localization 1350
Locals window 1317
local-variable declaration space 213
location in the computer's memory 75
Location property 487
lock keyword 600, 614, 615, 625
locking objects 599
Log in Web Form 1013
Log method of class **Math** 182
logarithm 182
logarithmic calculation 21
logging feature 1244
logic element 1247
logic error 73, 104, 109, 111, 250, 1313
logical AND operator (**&&**) 160, 162
logical decision 3
logical exclusive OR 163
logical negation (**!**) 160, 163
logical NOT operator (**!**) 160, 163
logical operator 21, 160, 163
logical OR operator (||) 160, 161, 162
logical unit 3
Login.aspx 1013
Login.aspx.cs 1016
loginUrl 1021
Logo language 277
720
long primitive data type 196
long promotions 194
long-term retention of data 757
longdesc attribute 1217
Look and Feel Observation 12
loop 109
loop body 144, 156
loop-continuation condition 141, 142, 143, 144, 146, 152, 156, 158, 159
loop counter 140
loop-terminating condition 250
loopback IP address 1119
looping process 117
Lovelace, Ada 11
low-order bit 1498
lowercase 62
lowercase letter 62, 73, 105
Lowest ThreadPriority enumeration member 594
lvalue 238
Lynx 1234

M

m-by-*n* array 265
machine dependent 6
machine language 6
MacOS operating system 6
Magenta static property of structure **Color** 689
magnetic disk 757
magnetic tape 757
mailto: URL 1382, 1439
Main method 63, 66, 72, 83, 108, 142, 581
Main thread of execution 595
main.html 1377, 1432
MainMenu and **MenuItem** properties and events 524
MainMenu class 522, 523
MainMenu properties 524
MainMenu properties and events 524
maintainability 1146
maintenance of software 12
making decisions 85
manipulating array elements 240
Manipulating the **Anchor** property of a control 487
"manufacturing" section of the computer 4
many-to-many relationship 905
map element 1416, 1477
marked for garbage collection 313
markup 950
markup language 28, 1375, 1431
Massachusetts Institute of Technology (MIT) 15
match 879
Match class 633, 669, 672
match the selection criteria 906
MatchCollection class 669

matching left and right braces 75
Math class 21, 150, 1096
Math class methods 181
Math tutor application 1098
math tutor application 1098
Math.Abs method 182
Math.Ceiling method 182
Math.Cos method 182
Math.E constant 181
Math.Exp method 182
Math.Floor method 182
Math.Log method 181, 182
Math.Max method 182, 193
Math.Min method 182
Math.PI constant 181
Math.Pow method 182
Math.Sin method 182
Math.Sqrt method 181, 182, 188
Math.Tan method 182
mathematical formula 839
Mathematical Markup Language (MathML) 25
MathML (Mathematical Markup Language) 25
Max method of class **Math** 182, 193
maxDigits attribute 1247
MaxDropDownItems property of class **ComboBox** 542, 543
MaximumSize property 488
maxlength attribute 1406, 1465
maxOccurs attribute 871, 873
maxTime attribute 1247, 1248
MaxValue constant of **Int32** 466
maze traversal 279
MBAFreeAgent.com 1300
MBCS (multi-byte character set) 1353
MDI (multiple document interface) 23, 565
MDI form 567
MDI parent and MDI child events and properties 567
MDI parent window and MDI child windows 566
Examples
    MDI parent-window class 571
MDI parent window class 571
MDI parent-window class 571
MDI title bar 567
MdiChildActivate event of class **Form** 567
MdiChildren property of class **Form** 567
MdiList property of class **MenuItem** 569
MdiParent property of class **Form** 567
mean (average) 78
MediaPlayerTest.cs 733
member 1338
member access modifier 284
member access modifier **private** 285
member access modifier **public** 284
member access operator 347
member tag 1346
Members column 1339
memory 3, 4, 13
memory consumption 1185
memory leak 312, 449
memory location 249
Memory location diagram 75
memory unit 4
MemoryStream class 760
menu 37, 69, 475, 521
menu access shortcut 522
menu access shortcut, create 523
menu bar 69, 71, 475
menu bar in Visual Studio .NET 37
Menu Designer in VS .NET 522
menu item 522
menu separator 523
<menu> tag 1241, 1242
menu, ellipsis convention 523
MenuItem properties 524
MenuItem property **MdiList** example 569
MenuItems property of class **MainMenu** 524
MenuItems property of class **MenuItem** 524
MenuMerge enumeration 568
MenuMerge.Add 568
MenuMerge.MergeItems 568
MenuMerge.Remove 568
MenuMerge.Replace 568
Menus used for changing text font and color 525
Menus used to change text font and color 525
MenuTest.cs 525
Mercury programming language 19
Merge records from Tables 912
MergeItems member of enumeration **MenuMerge** 568
MergeOrder property of class **MenuItem** 524, 568
MergeType property of class **MenuItem** 524, 568
Merlin **Microsoft Agent** character 736
Message 461
message box 21, 1313
message dialog 147
message dialog buttons 147
message dialog icons 147
Message property of class **Exception** 447, 451, 457
MessageBox class 68, 70, 243
MessageBoxButton.AbortRetryIgnore 148
MessageBoxButton.OK 148
MessageBoxButton.OKCancel 148
MessageBoxButton.RetryCancel 148
MessageBoxButton.YesNo 148
MessageBoxButton.YesNoCancel 148
MessageBoxButtons class 147
MessageBoxIcon class 147
MessageBoxIcon.Error 148
MessageBoxIcon.Exclamation 148
MessageBoxIcon.Information 148
MessageBoxIcon.Question 148
meta element 955, 1418, 1420, 1478, 1479
method 62, 63, 171, 179
method = "get" 1405, 1464
method = "post" 1405, 1464
method attribute 1248, 1405, 1463
method body 187
method call 180
method call operator 185
method call stack 457
method definition 63
method **Factorial** 216
method header 186
method overloading 223
MethodDebugExample.cs 1322

microprocessor chip technology 13
Microsoft 1351
Microsoft Agent 24, 736, 742
Microsoft Agent Character Editor 736
Microsoft Agent Control 2.0 742
Microsoft Agent demonstration 742
Microsoft Intermediate Language (MSIL) 18, 66
Microsoft Internet Explorer accessibility options 1262
Microsoft Linguistic Sound Editing Tool 736
Microsoft **Magnifier** 1251
Microsoft **Narrator** 1258, 1261
Microsoft .NET 16
Microsoft **On-Screen Keyboard** 1261, 1262
Microsoft SansSerif font 697
Microsoft Schema file that contains structure to which `bookxdr.xml` conforms. 871
Microsoft Serif font 697
Microsoft SQL Server 897
Microsoft Windows 95/98 60
Microsoft Windows NT/2000 60
middle array element 261
middle tier 953
MIDI (Musical Instrument Digital Interface) 733
`Millimeter` member of enumeration `GraphicsUnit` 696
MIME (Multipurpose Internet Mail Extensions) 952, 988
`Min` method of class `Math` 182
minimized and maximized child window 568
Minimized and maximized child windows 568
`MinimumSize` property 488
`minOccurs` attribute 871, 873
minus sign (`-`) 841
MIT (Massachusetts Institute of Technology) 15
MIT's Project Mac 13
mixed-type expression 194
modal dialog 776
`mode` attribute 1021
model attribute 871
modifier key 511
modify a variable at run time 1317
Modifying a database 930
modularizing a program with methods 182
module 179
modules in C# 179
modulus assignment operator (`%=`) 121
modulus operator (`%`) 76, 77
`Monitor` class 592, 594, 595, 599, 607, 614, 615
monolithic excecutable 1362
`Monster.com` 1290, 1294, 1298, 1300
Moore's Law 13
More Windows... option in Visual Studio .NET 569
Morse code 1144
Motion Pictures Experts Group (MPEG) 733
mouse 3, 475
Mouse Button Settings 1258
mouse click 509
mouse cursor 70, 1253
mouse event 509
Mouse events, delegates and event arguments 509
mouse move 509
mouse pointer 39, 40, 70
mouse press 509
Mouse Speed dialog 1259
mouse, drag 42
`MouseDown` event 509
`MouseEventArgs` class 509
`MouseEventArgs` properties 509
`MouseHover` event 509, 1226
MouseKeys 1257
`MouseLeave` event 509
`MouseMove` event 509
`MouseUp` event 509
`Move` method of class `Directory` 762
`Move` method of class `File` 761
`MoveNext` of `IEnumerator` 1194
`MoveTo` method of interface `IAgentCtlCharacter` 750
MPEG format 733
MS-DOS prompt 60
`mscorlib.dll` 69
MSDN documentation 147
`msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp` 868
MSIL (Microsoft Intermediate Language) 18, 66
`MSN.com` 987
msxml parser 841
multi-tier application 952
multi-byte character set (MBCS) 1353
multicast delegate 426
multicast event 479, 484
`MulticastDelegate` class 426, 479
`MultiColumn` property of class `ListBox` 535
multidimensional array 237, 265
`MultiExtended` value of `SelectionMode` 534
multilevel priority queue 594
`MultiLine` property of class `TabControl` 489, 561
multimedia 733
multiple document interface (MDI) 23, 565
multiple inheritance 343
multiple-subscripted array 265, 266
multiple-line comment (`/* */`) 61
multiple-selection structure 98, 152, 170
multiplication assignment operator (`*=`) 121
multiplicative operators: `*`, `/` and `%` 115
multiprogramming 5
Multipurpose Internet Mail Extensions (MIME) 952, 988
`MultiSelect` property of class `ListView` 553, 554
`MultiSimple` value of `SelectionMode` 534
multitasking 11
multithread safe 1150
multithreading 11, 24, 591
Musical Instrument Digital Interface (MIDI) 733
mutual exclusion 501
mutually exclusive options 501
MySQL 897

N

`N` 151
n-tier application 952
`name` attribute 873, 1405, 1465
`name` attribute of `member` tag 1346
`name` attribute of `meta` element 955
`name` node-set function 881
name of a control variable 140
name of a variable 75
name of an attribute 1433
`Name` property of class `Font` 493, 696
`Name` property of `HttpCookie` class 996, 997
`Name` property of structure `Color` 693
named constant 242
namespace 62, 68, 284, 288, 844
`namespace` keyword 328
namespace prefix 844, 847
namespace prefix `xsd` 873
`Namespace` property of a `WebService` attribute 1051
`namespace.xml 844`, 844
`NamespaceURI 856`
naming collision 328, 844
`NaN` constant of class `Double` 464
Narrator reading **Notepad** text 1261
narrowing conversion 193
natural logarithm 182
`nav.html` 1384, 1443
`NavigateUrl` attribute 974
navigation bar 1015
navigational frame 1420, 1480
negative arc angles 705
negative infinity 446
`NegativeNumberException.cs` 463
nested tags 1422
nested block 213
nested building block 170
nested control structure 116, 155, 168
nested element 840, 1434
nested `for` loop 243, 267, 268, 269
nested `frameset` element 1424, 1483, 1484
nested `if` structure 103
nested `if/else` structure 102, 265
nested list 1389, 1391, 1448
nested parentheses 77
nested within a loop 117
nesting 171
nesting rule 168
.NET initiative 16
.NET-compliant language 19
.NET component 1363
.NET Framework 18, 180
.NET Framework Class Library 27, 62, 79, 179, 591
.NET Languages 19
.NET platform 9
Netscape Communicator 1431
network address 1108
network message arrival 440
networking 757
`NetworkStream` class 1110
`new` operator 239, 287, 311, 1148, 1185
New Project dialog 36
new project in Visual Studio .NET 38
New `Text` property value reflected in design mode 129
`NewDialog.cs` 827
`NewDialogForm` class enables users to create records in transaction-processor case study 827
newline character (`\n`) 62, 67, 73
News article formatted with XML 839
`NewValue` event of class `CheckedListBox` 541
`next` attribute of `choice` element 1241
`Next` method of class `Random` 201, 1097
`NextNode` property of class `TreeNode` 549
no-argument constructor 293, 295
node 547
node, child 547
node, expand and collapse 547
node, parent 547
node, root 547
`Nodes` property of class `TreeNode` 549
`Nodes` property of class `TreeView` 548
node-set function 881
`noframes` element 1422, 1423, 1482
nondestructive 76
`None` value of `SelectionMode` 534
nonfatal logic error 104
nonlinear data structures 1148
nonprimitive data type array 250
nonrecursive method call 223
nonvalidating XML parser 865
NOT operator (logical) 160, 163
not-selected state 501
Notepad text editor 530, 1376, 1431
noun 11
`Now` property of structure `DateTime` 580
n-tier application 6
`null` reference 212, 1147
`NullReferenceException` 448
Number systems Appendix 1275

O

Oberon programming language 19
object 8, 11, 281, 282
object-based programming (OBP) 2, 282
Object Browser (Visual Studio .NET) 333
`Object` class 284, 401, 642
Object debugging example 1325
object of a derived class 383
object of a derived class is instantiated 371
object orientation 281
object oriented 282
object-oriented programming (OOP) 2, 8, 11, 282, 343, 383
object passed by reference 282
`object` primitive data type 197
"object speak" 281
"object think" 281
object-oriented programming 844
objects constructed “inside out” 377
OBP (object-based programming) 282
occurence indicator 866
octal (base8) 176
Ocularis 1250
`.OCX` file 1363, 1364
”off-by-one error” 143, 239
`OK` 148
OK button on a dialog 70
`OKCancel` 148
`ol` (ordered list) tag 1392

OleDbCommand class 920
OleDbConnection class 920
OleDbDataAdapter class 920
OleDbDataReader class 1020
one-based counting 143
one comparison in the binary search algorithm 262
one-dimensional array 265
one statement per line 84
one-to-many relationship 904
One value of **SelectionMode** 534
one's complement (**~**) 1283, 1496, 1504
ones position 1277
onHangup element 1247
OnInit method 957
online contracting service 1300
online guest book 1006
online recruiting 1292
onMaxSilence element 1247
OnPaint method from class **Control** 578, 579, 687
On-Screen Keyboard 1261
onTermDigit element 1247
OOP (object-oriented programming) 2, 8, 11, 282, 343, 383
Open method of class **File** 761
open-source software 5
open technology 839
opened 760
OpenFileDialog class 783, 788
opening a project 38
OpenRead method of class **File** 761
OpenRead method of class **WebClient** 1085
OpenText method of class **File** 761
OpenWrite method of class **File** 761
operand 74
operating system 5
operations of an abstract data type 327
operator **=** 74
operator **==** 83
operator keyword 432
operator overloading 430
operator precedence 77
operator precedence chart 27, 115
Operator precedence chart Appendix 1273
Operators **checked** and **unchecked**, and handling arithmetic overflow 467
optical disk 757
optimization 151
OptionPage.aspx.cs 991
Options supplied on an ASPX page 989, 997
OptionsPage.aspx 989, 997
OptionsPage.aspx.cs 999
Or method of class **BitArray** 1508
OR operator (boolean logical exclusive) 160
OR operator (boolean logical inclusive) 160
OR operator (logical) 160, 161
Oracle 897
Oracle Corporation 1351
Orange static property of structure **Color** 689
order attribute 882
ORDER BY 905, 909, 910, 911
Order in which constructors and destructors are called 376
ordered 897
ordered list 1448, 1450
ordered list element 1392
ordering of records 905
out keyword 198
out-of-line transfer of control 403
out-of-range array subscript 440, 448
Out property of class **Console** 759
outer block 213
outer **for** structure 259, 269
outer set of square brackets 249
OutOfMemoryException 1148
output 37, 64, 67
output cursor 64, 66
output device 4
output directory 581
output file 581
output unit 4
oval symbol 98
overflow 440, 466
Overflow.cs 467
OverflowException class 466
overhead of recursion 222
overlapped building block 170
overload resolution 224
overloaded constructor 292
Overloaded constructor demonstration 295
Overloaded constructors provide flexible object-initialization options 293
overloaded method 223
overloading 223, 366
overridden 344
override keyword 393
Oz programming language 19

P

p (paragraph) element 1378, 1434
packet 1108
Page class 957, 983, 999
  Session property 999
  trace property 1028
page content 1378
page layout software 633
page tracing 1027
Page_Unload method 958
<%@Page…%> directive 954
PageIndexChange event 1027
PageIndexChange event for a **DataGrid** control 1027
pageLayout property of ASP.NET page 963
PagerStyle property 1027
PagerStyle property of a **DataGrid** control 1027
PageSize property 1027
PageSize property of a **DataGrid** control 1027
Paint 559
Paint Shop Pro 1441
Painter.cs 510
PaintEventArgs class 578, 687
PaintEventArgs properties 579
palette 47
palindrome 278
Palo Alto Research Center (PARC) 8
panel 476
Panel class 495, 686
Panel Controls property 495
Panel properties 495
Panel properties and events 495
panel with scrollbars 496
parallelogram 344
param tag 1338
parameter 181, 185
parameter-list 186

PARC (Palo Alto Research Center) 8
parent element 841
parent menu 522
parent node 547, 847, 1169, 1209
parentheses **()** 77
parentheses "on the same level" 78
Parse method 75, 114
parsed character data 867
parser 841, 846
partition 279
partitioning step 278, 279
Pascal programming language 10, 19
Pascal, Blaise 10
pass-by-reference 197
pass-by-value 197
pass of a sorting algorithm 257
passing an array element 250
Passing an array reference using **ByVal** and **ByRef** with an array 254
passing an array to a method 250
Passing arrays and individual array elements to methods 251
password box 1406, 1466
password textbox 488
PasswordChar property of **TextBox** class 488, 489
paste 38
Path property of **HttpCookie** class 997
path to a resource 950
PathGradientBrush class 711
PathNavigator.cs 858
Paths used to draw stars on a form 715
pattern matching 908
pattern of **1**s and **0**s 758
payroll system 390, 758
#PCDATA flag 867
Peedy calculating the total 742
Peedy flying animation 739
Peedy introducing himself when the window opens 737
Peedy recounting the order 741
Peedy repeating the user's request for anchovies as an additional topping 741
Peedy repeating the user's request for Seattle-style pizza 740
Peedy the Parrot**Microsoft Agent** character 736
Peedy waiting for speech input 740
Peedy's reaction when he is clicked 738
Peek method of class **Stack** 1198
Pen class 685, 689, 701
percent sign (**%**) is the modulus operator 76
Performance Tip 12
performing a calculation 85
Perl progamming language 19
permission settings 554
persistent data 757
persistent information 987
Person class implements **IAge** interface 415
personal computer 3
personal computing 5
personalization 987
PhotoShop Elements 1441
PhysicalApplication-Path property of **Request** class 1006
"pick off" each digit 93
picture box 49, 507
picture.html 1382, 1441
PictureBox class 507, 570
PictureBox properties and events 507
PictureBoxTest.cs 507
PieceWorker class inherits from class **Employee** 408
Pig Latin 683
pin a window 40
Pink static property of structure **Color** 689
pixel 1441
Pixel member of enumeration **GraphicsUnit** 696
platform independence 17, 19
play element 1249
Play method of interface **IAgentCtlCharacter** 748
playback, choppy 592
player thread 592
plus sign (**+**) 841
plus sign (**+**) occurence indicator 866
Point class represents an *x*-*y* coordinate pair 348, 384
Point marked up with XML comments 1331
point-of-sale system 794
Point structure 686
Point2 class inherits from abstract class **Shape** 395
Point2 class represents an *x*-*y* coordinate pair as **protected** data 357
Point3 class implements interface **IShape** 420
Point3 class uses properties to manipulate its **private** data 362
Point3.cs 420
Point4 base class contains constructors and finalizer 372
PointTest class demonstrates class **Point** functionality 350
poker 683
Polygon-drawing demonstration 708
polymorphic processing of related errors 448
polymorphic programming 390, 394
polymorphic screen manager 391
polymorphism 22, 156, 282, 378, 383, 388, 391, 393, 403
polymorphism as an alternative to **switch** logic 437
polynomial 79, 80
pool of threads 1110
Pop method of class **Stack** 1194, 1195
pop stack operation 1160
popping off a stack 326
port number 1108, 1109
portability 19, 1353
Portability Tip 12
portable 839
porting 19
position number 237
positional notation 1277
Positional value 1278
Positional values in the decimal number system 1278
705
positive and negative arc angles 705
positive infinity 446
post request type 1248, 1464
postback 983
postdecrement 121, 145
postincrement 121, 123, 145
postorder traversal of a binary tree 1170

Pow method of class **Math** 150, 151, 182, 1096
power 182
Precedence and associativity chart 166
Precedence and associativity of the operators discussed so far in this book 123, 239
precedence chart 78, 115
precedence chart Appendix 1273
precedence of arithmetic operators 78, 123, 239
precedence rule 77
predecrement 121
predicate method 285, 1150
preincrement 121, 123, 145
premature program termination 249
preorder traversal of a binary tree 1170
prepackaged data structures 1185
presentation logic 953
presentation of a document 1375, 1431
previous statement, variable in 1317
PrevNode property of class **TreeNode** 549
primary interop assembly 1367
primary key 897, 904
primary memory 4
prime number 1508
primitive (or built-in) data-type 194, 196, 212
primitive data type 73
primitive data types are initialized to null 212
Princeton Review 1302
print a line of text 64
print spooling 1165
PrintBits.cs 1498
printing a project 38
Printing on multiple lines with a single statement 67
Printing on one line with separate statements 67
Priority property of class **Thread** 595
privacy invasion 987
privacy protection 987
private member access modifier 284, 285, 291, 297
Private members of a base class 346
probability 201
procedural programming language 11, 282
procedure for solving a problem 95
Process class 531
processing instruction 878
processing instruction target 878
processing instruction value 878
processing phase 112
processing unit 3
produce method 601
producer 615
Producer and consumer threads accessing a circular buffer 617
producer and consumer threads accessing a circular buffer 617
Producer and consumer threads accessing a shared object with synchronization 607
producer and consumer threads accessing a shared object with syncronization 607
Producer and consumer threads accessing a shared object without synchronization 602
producer and consumer threads accessing a shared object without syncronization 602
producer thread 601
producer/consumer relationship 601
productivity 13
program 3
program construction principles 140
program control 60, 96
program development 60
program development process 326
program development tool 120
program execution stack 1160
program in the general 437
program termination 249
Program that demonstrates class **Array** 1186
Program that prints histograms 243
Program to display hidden text in a password box 490
Program to simulate the game of craps 208
program, break execution 1318
program, suspend 1313
programmer 3
programmer-defined class 62
Examples
  Programmer-defined control that displays the current time 580
Programmer-defined control that displays the current time 580
Programmer-defined **Maximum** method 188
programmer-defined method 180
programmer-defined type 282
project 36
Project Mac 13
Project menu of VIsual Studio 38, 283
Project properties dialog 582
project, Windows control library 581
promotion 115, 987
prompt 73, 114
prompt element in VoiceXML 1241
<prompt> tag 1242
Properties demonstration for class **Time3** 301
Properties of class **Control** related to accessibility 1225
Properties window 42, 43, 46, 49, 126, 963
Properties window on a Web Page 1028
Properties.cs 458
property 42
property definition 291
property for a form or control 42
property of an object 11, 20
Proposed Recommendation 15
Protected 347
protection 1021
proxy 1364
proxy class for Web services 1046, 1053, 1054
pseudocode 21, 96, 99, 105, 107, 110, 111, 118
pseudocode algorithm 111
pseudocode **if/else** structure 101
pseudocode statement 105
pseudo-random number 201
public interface 285
public keyword 405
public member access modifier 284, 285
public member of a derived class 346
public method 286

public operation encapsulated in an object 290
public service 285
public static members 313
Publication page of Deitel and Associates’ VoiceXML page 1238
publishers table of **books** database 898, 899
publishing a Web service 1046
Pulse method of class **Monitor** 594, 600, 607
Pulse method of **Monitor** 615
PulseAll method of class **Monitor** 594, 600
Push method of class **Stack** 1194
push stack operation 1160
pushing into a stack 326
PWS (Personal Web Server) 1405
Pythagorean Triples 177
Python progamming language 19

Q

quantifier 671
Quantifiers used in regular expressions 672
Quantifiers used regular expressions 672
quantum 594
quantum expiration 593
query 896, 898
query a database 921
Question 148
question mark (**?**) occurence indicator 866
queue 26, 285, 327, 1146, 1165
Queue class 1185
QueueInheritance extends class **List** 1166
QueueInheritanceLibrary.cs 1166
QueueTest.cs 1167
quicksort 278

R

R property of structure **Color** 689
RAD (rapid applications development) 9, 23, 327
radian 182
radio 1409, 1469
radio button 488, 501
radio button group 501
radio buttons, using with **TabPage** 565
RadioButton class 498, 501
RadioButton properties and events 502
RadioButtonsTest.cs 502
RadioCheck property of class **MenuItem** 524, 530
RAM (Random Access Memory) 4
random-access file 25, 757, 794, 807, 827
random access memory (RAM) 4
Random class 200, 1097
random number generation 664, 683
Random-access file with fixed-length records 795
RandomAccessRecord.cs 795
rapid application development (RAD) 9, 23, 327
rapid applications development (RAD) 327
RCW (Runtime Callable Wrapper) 1368
RDBMS (relational database management system) 952
RDK (Redistribution Kit) 1249
Read method 876
Read method of class **Console** 760
Read method of class **NetworkStream** 1110
read-only text 488
readability 61, 62, 1218, 1266, 1432
ReadByte method of class **NetworkStream** 1110
807, 784
ReadLine method of class **Console** 73, 760
readonly keyword 22, 317
ReadOnly property 489
807, 784
ReadString method of class **BinaryReader** 1119
ReadXml method of **DataSet** 938
Ready thread state 593
real number 73
receive a connection 1118
receive data from a server 1119
Receive method of class **Socket** 1109
Receive method of class **UdpClient** 1120, 1125
ReceiveFrom method of class **Socket** 1109
receiving an array through a method call 250
“receiving” section of the computer 4
receiving temperature and weather data from a Web service 1087
recent project 34
reclaim memory 316
reclaiming dynamically allocated memory 592
recognizing clients 987
RecommendationsPage.aspx 994, 1003
RecommendationsPage.aspx.cs 995, 1004
record 758, 897, 903
Record for random-access file-processing applications 795
Record for sequential-access file-processing applications 774
record key 758, 794
record set 898
record size 794
Record.cs 774
recordAudio element 1249
Record-transaction class for the transaction-processor case study 812
Recruitsoft.com 1297
Rectangle structure 685, 686, 702, 714
rectangle symbol 98, 100, 105, 145, 155, 166
rectangular array 265
rectangular hotspot 1418, 1477
recursion 553
recursion overhead 222
recursion step 216, 220
recursion vs. iteration 222
recursive call 216, 220, 221
Recursive evaluation of 5! 217
recursive method 21, 215, 218
recursive method **Factorial** 216
recursive program 221
recursive searching 278
recursive step 278
recursive version of the binary search 260

recursive version of the linear search 260
Recursively generating Fibonacci numbers 219
red circle, solid 1314, 1319
`Red` `static` property of structure `Color` 689
Redistribution Kit (RDK) 1249
redundant parentheses 79
`ref` keyword 198
reference 1338
reference manipulation 1146
reference to a new object 287
reference type 196
referring to a base-class object with a base-class reference 389
referring to a base-class object with a derived-class reference 390
referring to a derived-class object with a base-class reference 390
referring to a derived-class object with a derived-class reference 389
refinement process 110
`Regex` class 633, 669, 670
`Regex` methods `Replace` and `Split` 677
`Regex` methods `Replace and` `Split` 677
`RegexMatches.cs` 669
`RegexSubstitution.cs` 677
regional invalidation 722
register event handler 483
`<%@Register…%>` directive 1015
Registering an ActiveX control 1363
`RegSvr32` utility 1363, 1367
regular expression 765
766
Regular expressions checking birthdays 669
`Regular` member of enumeration `FontStyle` 697
“reinventing the wheel” 62
relational database 896
relational database management system (RDBMS) 952
relational database model 897
relational database table 897
Relational-database structure of an `Employee` table 897
relative positioning 963
release a lock 614, 615
release resource 450
release the lock 626
remainder 77
`remarks` tag 1338
remote machine 1041
Remote Procedure Call (RPC) 16, 1041
`Remove` member of enumeration `MenuMerge` 568
`Remove` Method if `TreeNodeCollection` 853
`Remove` method of class `ArrayList` 1189, 1193
`Remove` method of class `StringBuilder` 658
`Remove` method of class `TreeNodeCollection` 549
`Remove` method of `Hashtable` 1204
`RemoveAt` method of class `ArrayList` 1189
`RemoveRange` method of class `ArrayList` 1189
Removing tabs from Visual Studio environment 1223
renaming code files in Visual Studio .NET 66
renders 952
repetition 167, 170, 171
repetition control structure 97, 98, 99, 105, 110
repetition structure 21
repetition structure for iteration 222
`Replace` member of enumeration`MenuMerge` 568
`Replace` method of class `Regex` 677, 678
`Replace` method of class `String` 648, 649
`Replace` method of class `StringBuilder` 659
Report Expert 1514
`Request` class 1006
  `Cookies` property 996
  `PhysicalApplicationPath` property 1006
request for proposal 1300
`Request` object 983, 996
`RequiredFieldValidator` class 979
`Reservation.asmx.cs` 1076
reserved word 62
reserving memory 240
`Reset` of `IEnumerator` 1194
resolution 686
resource leak 311, 441, 449
`Response.Write` 1027
responses to a survey 247, 249
Restart button 1318
result of an uncaught exception 442
result set 898, 906
result tree 877
Results of combining two bits with the bitwise AND operator (`&`) 1497
Results of combining two bits with the bitwise exclusive OR operator (`^`) 1498
Results of combining two bits with the bitwise inclusive OR operator (`|`) 1497
results of invoking a Web service method from a Web browser 1044
resume 1291, 1296
`Resume` method of class `Thread` 594
resume-filtering software 1296
resumption model of exception handling 442
rethrow an exception 456
`RetryCancel` 148
return from a method 180
`return` statement 185, 186, 188, 216
return-value-type 186
returning an object from a Web-service method 1097
`returns` tag 1338, 1339
reusability 1146
reusable component 344
reusable software component 12, 13, 40, 328
reuse 62
`Reverse` method of class `Array` 1188
RGB values 688, 689
Richards, B. Martin 7
right brace (`}`) 63, 71, 114
right child 1169
right-shift assignment operator (`>>=`) 1508
right-shift operator (`>>`) 1496, 1497, 1504
right subtree 1169, 1176
`RightToLeft` property of class `MainMenu` 524

"rise-and-shine algorithm" 95
Ritchie, Dennis 7
Robby the Robot **Microsoft Agent** character 736
robust application 74, 439
Rolling dice in a windows application 203
root element 840, 868
root node 547, 847, 1169
root node, create 547
RotateTransform method of class **Graphics** 717
round 182
Round member of enumeration **DashCap** 715
round-robin 594
rounding 77
rounding error 195
row 897
rows attribute (**textarea**) 1408, 1465
rows to be retrieved 905
rowspan attribute (**tr**) 1403, 1460
RPC (Remote Procedure Call) 16, 1041
RPG progamming language 19
Rule of Entity Integrity 903, 904
Rule of Referential Integrity 904
rules of operator precedence 77
Run command in Windows 531
run element 1248
Run menu 52
run mode 52
run-time exception 448
run-time logic error 73
Runnable thread state 593
running an application 531
Running thread state 593, 596
Runtime Callable Wrapper (RCW) 1368

S

Salary.com 1302
783
Sample Internet Explorer window with GUI components 476
SaveFileDialog class 776
Saving a document to a file 1340
sbyte primitive data type 196
sbyte promotions 194
scalar quantities 250
scalars 250
scaling 201
scaling factor 201
scheduling 594
Schema 865, 869
schema element 873
schema repository 871
Schemas property of **XmlSchemaCollection** 876
Schema-validation example 874
Scheme progamming language 19
scope 212, 290
scope (block) 213
scope (class) 213
scope (global) 213
scope of a variable 144
scope of an identifier 212
Scoping example 213
screen 3, 4, 5
screen cursor 68
screen-manager program 391
screen reader 1217, 1230, 1248, 1258, 1263
script 1357, 1433
scrollbar 475, 476
scrollbar and window border size dialog 1252
scrollbar in panel 496
SDI (single document interface) 565
SDI and MDI forms 566
SDK (Software Development Kit) 1249
sealed 402
search engine 1378, 1418, 1434, 1478
search key 260
searching 259, 260, 1146
Searching for characters and substrings in **String**s 643
searching technique 237
second-degree polynomial 79, 80
second refinement 117
secondary storage 4, 13
secondary storage device 757
sector 705
Secure property of **HttpCookie** class 997
secure protocol 997
seed 201
SeekOrigin enumeration 806
SeekOrigin.Begin constant 806
SELECT 898, 905, 906, 907, 908, 909, 910, 911, 913
select 898, 905
select all fields from a table 906
Select Case logic 390
SelectCommand property of **OleDbAdapter** 927
selected attribute 1473
selected state 501
SelectedImageIndex property of class **TreeNode** 549
SelectedIndex property of class **ComboBox** 543
SelectedIndex property of class **ListBox** 535
SelectedIndex property of class **TabControl** 562
SelectedIndexChanged event of class **ComboBox** 543
SelectedIndexChanged event of class **ListBox** 536
SelectedIndexChanged event of class **TabControl** 562
SelectedIndices property of class **ListBox** 535
SelectedItem property of class **ComboBox** 543
SelectedItem property of class **ListBox** 535
SelectedItem property of class **ListView** 554
SelectedNode property of class **TreeView** 549
SelectedTab property of class **TabControl** 562
selecting 34
Selecting the **Build Comment Web Pages** from **Tools** menu 1339
selection 167, 168, 170
selection control structure 21, 97, 98, 99
selection criteria 906
selection structure for recursion 222
SelectionMode enumeration 534
SelectionMode property of class **CheckedListBox** 540
SelectionMode property of class **ListBox** 534, 535
SelectionMode.MultiExtended 534
SelectionMode.MultiSimple 534
SelectionMode.None 534
SelectionMode.One 534

self-documenting 73
self-referential class 1146, 1148
Self-referential class objects linked together 1148
Self-referential object 1147
semicolon (**;**) 64, 73, 84, 104
send data to a server 1119
Send method of class **Socket** 1109
Send method of class **UdpClient** 1120, 1125
sendEvent element 1248
SendTo method of class **Socket** 1109
sentinel-controlled repetition 109, 110, 111, 114
sentinel value 109, 114
separator bar 523
separator, menu 523
sequence 167, 170, 171, 1168
sequence control structure 97, 99, 110
sequence element 873
sequence of items 1148
sequence structure 21
sequence type 1450
sequential-access file 757, 759, 776, 794, 827
sequential execution 96
sequential file 25
Serializable attribute 771
SerializationException 784
Serialize method of class **BinaryFormatter** 782
serialized object 771
server 6
server Internet address 1119
server port number 1119
Server portion of a client/server stream-socket connection 1111
Server side of client/server Tic-Tac-Toe program 1126
Server.cs 1111, 1120
Server-side portion of connectionless client/server computing 1120
ServerValidate event of **CustomValidator** class 1015, 1020
ServerValidateEventArgs class 1020
  IsValid property 1020
  Value property 1020
service 291
service description for a Web service 1042, 1043
session 1243
session attribute 1248
Session data read by an ASP .NET Web application to provide recommendations for the user 1004
session ID 987
Session information displayed in a **ListBox** 1003
session item 1002
Session property of **Page** class 999
session tracking 987
sessionID 1243
SessionID property of **HttpSessionState** class 1003
Sessions are created for each user in an ASP .NET Web application 999
Sessions created for each user in an ASP .NET Web application 999
session-tracking 988
set accessor 291, 297, 298, 301
Set Automatic Timeouts 1258, 1259
SET keyword 918
SET keyword 918
Set method of class **BitArray** 1508
Set of recursive calls to method **Fibonacci** 221
SetAll method of class **BitArray** 1510
setAttribute method of interface **HttpSession** 1002
SetAuthCookie method of **FormAuthenication** class 1020
SetDataBinding method of **DataGrid** 928
Setting the form's **Text** property 46
Setting the project location 46
SGML (Standard Generalized Markup Language) 14
shape class hierarchy 346, 347, 381
Shapes 712
Shapes drawn on a form 712
shared buffer 601, 615
shared library 1362
shared memory 601
shift 201
Shift key 511
Shift property 514
Shifted random integers 201
"shipping" section of the computer 4
short primitive data type 196
short promotions 194
short-circuit evaluation 162
shortcut key 522, 1221
Shortcut key creation 1223
Shortcut property of class **MenuItem** 524
shortcuts with the **&** symbol 522
Show method 485
Show method of class **Form** 567, 574
Show method of class **MessageBox** 70, 147
Show method of interface **IAgentCtlCharacter** 748
Show Next Statement button 1318
Show of class **Form** 478
690
ShowDialog method of class **OpenFileDialog** 783, 788
ShowDialog method of class **SaveFileDialog** 776
ShowShortcut property of class **MenuItem** 525
ShowSounds 1255, 1257
sibling 1169
sibling node 547, 847
side effect 162, 221
Sieve of Eratosthenes 1508
signal value 109
signature 224, 225, 366
silicon chip 3
Simple Class Library 332
simple condition 160
Examples
  Simple event-handling example using visual programming 481
Simple event-handling example using visual programming 481
Simple Object Access Protocol (SOAP) 10, 16, 20, 1041, 1043
Simple program as it executes 45

Simple student-poll analysis program 248
Simple style for **ComboBox** 544
SimpleEventExample.cs 481
simplest flowchart 168, 169
Simula 67 programming language 8
simulate coin tossing 234
Simulating rolling 12 six-sided dice 205
simulation 200
Sin method of class **Math** 182
sine 182
single document interface (SDI) 565
single-entry/single-exit control structure 99, 100, 166, 167
single inheritance 343
single-line comment 61, 73, 75
single-quote character (**'**) 844
single-selection structure 98, 170
single-subscripted array 265, 266
singlecast delegate 426
single-clicking with left mouse button 34
single-quote character 908
sinking a large value 259
SixFigureJobs 1301
size attribute (**input**) 1406, 1465
size of a variable 75
Size property of class **Font** 488, 696
SizeInPoints property of class **Font** 696
SizeMode property 507
sizing handle 47
sizing handle, disabled 47
sizing handle, enabled 47
Sleep method of class **Thread** 594, 595, 606
sleeping thread 595
small circle symbol 98, 155
SmallImageList property of class **ListView** 554
Smalltalk programming language 8, 19
SMIL (Synchronized Multimedia Integration Language) 25, 1250
"sneakernet" 6
SOAP (Simple Object Access Protocol) 10, 16, 20, 1041, 1043
SOAP encoding rule 16
SOAP envelope 1045
SOAP request 1045
SOAP Request for the **HugeInteger** Web service 1045
socket 1107
Socket class 1118
software 3
software component 17
Software Development Kit (SDK) 1249
Software Engineering Observation 12
software reusability 12, 182, 290, 328, 343
Solaris 6
solid arc 705
solid polygon 708
solid rectangle 702
SolidBrush class 693, 697, 701
solution 36
Solution Explorer after adding a Web reference to a project 1057
Solution Explorer in Visual Studio .NET 39, 574
Solution Explorer window 40, 1367
Solution Explorer window for project **WebTime** 961
solution, debug setting 1314
Some basic GUI components 476
Some common escape sequences 68
Some methods of class **ArrayList** 1189
Sort method of class **Array** 1185
Sort method of class **ArrayList** 1189
Sort property 1027
Sort property in **DataView** class 1027
sorted array 1150
Sorted property of class **ComboBox** 543
Sorted property of class **ListBox** 535
SortedList class 1185
sorting 257, 1146
sorting a large array 259
Sorting an array with bubble sort 257
sorting schemes 257
sorting technique 237
sorting.xml 878
sorting.xsl 879
SoundSentry 1254
source code 1339
source-code form 1376, 1431
source tree 877
space character 62
spacing convention 63
span attribute 955, 1460
spawning 1109
special character 62, 634, 1387, 1405, 1445, 1446
special symbol 758
speech device 1459
speech recognition 24, 1250, 1265
speech recognition engine 736
speech synthesis 24, 1250, 1265
speech synthesizer 1250, 1442
spiral 219
split a statement 70
Split method of class **Regex** 677, 678
spooling 1165
sports.xml 864
SQL (Structured Query Language) 896, 898, 905
SQL keywords 905
SQL query keywords 905
SQL statement 898
Sqrt method of class **Math** 182, 188, 464
square 344, 1450
square brackets in a query 906
square root 182
SquareRootTest.cs 464
src attribute (**img**) 1385, 1441, 1444
Src file 1015
stack 26, 234, 285, 326, 1160
stack 1160
Stack class 1185, 1194
stack unwinding 443, 458
StackComposition class encapsulates functionality of class **List** 1164
StackCompositionLibrary.cs 1164
stacking 171
stacking rule 168
StackInheritance extends class **List** 1161
StackInheritanceLibrary.cs 1161
StackInheritanceTest.cs 1162
StackTest.cs 1195
StackTrace property of **Exception** 457, 458, 461

standard character 1405
standard error 760
Standard Generalized Markup Language (SGML) 14
standard input 759
Standard ML language 19
standard output object 64, 760
standard reusable component 344
standard time format 288
start a multiline comment (`/*`) 61
Start button 52
`Start` method of class `Process` 531
`Start` method of class `TcpListener` 1109
`Start` method of class `Thread` 593, 598
Start Page 34
Start Page in Visual Studio .NET 35
`start` tag 598, 840, 843, 1433
`StartDialog.cs` 817
`StartDialogForm` class enables users to access dialog boxes associated with various transactions 817
Started thread state 593, 615
starting angle 704
`StartsWith` and `EndsWith` methods 641
`StartsWith` method of class `String` 641
startup project 39
starvation 595
state button 498
stateless protocol 987
statement 64
statement terminator (`;`) 64
`static` constructor 312
static duration 212
static entities 237
`static` keyword 312, 313, 315, 316, 317, 318
`static` member demonstration 315
`static` members are accessible to all objects of a clas 314
`static` method 70, 150
`static` method cannot access non-`static` class members 313
`Static` method `Concat` 648
`static` variable 313
Step Into button 1323
Step Out button 1323
Step Over button 1318
StickyKeys 1255
Stop Debugging button 1318, 1320
Stopped thread state 593, 594, 627
straight line 715
straight-line form 77
`Stream` class 760
stream input/output 757
stream of bytes 759
stream of input 109
stream socket 1125
`StreamReader` class 760
streams 1108
streams-based transmission 1120, 1140
`StreamWriter` class 760
`Strikeout` member of enumeration `FontStyle` 697
`Strikeout` property of class `Font` 696
`string` 24, 64
`String` class 286, 405, 406, 633
String Collection Editor in Visual Studio .NET 536
`string` concatenation 83
string constant 634
`String` constructors 635
`string` formatting codes 151
`String` indexer 638
`String` indexer, `Length` properties and `CopyTo` method 636
`String Length` property, the `CopyTo` method and `StrReverse` function 636
string literal 64, 634
`String` methods `Replace`, `ToLower`, `ToUpper`, `Trim` and `ToString` 649
string of characters 64
`string` primitive data type 72, 197
`String` test to determine equality 639
`String` testing for equality 639
`StringBuilder` class 633, 651
`StringBuilder` class constructors 651
`StringBuilder` size manipulation 653
`StringBuilder` text insertion and removal 658
`StringBuilder` text replacement 659
`StringBuilder`'s `AppendFormat` method 656
`StringBuilderAppend.cs` 655
`StringBuilderAppendFormat.cs` 656
`StringBuilderConstructor.cs` 651
`StringBuilderFeatures.cs` 653
`StringBuilderInsertRemove.cs` 658
`StringBuilderReplace.cs` 659
`StringCompare.cs` 639
`StringConstructor.cs` 635
`StringHashCode.cs` 642
`StringIndexMethods.cs` 643
`StringMethods.cs` 636
`StringMiscellaneous2.cs` 649
`StringStartEnd.cs` 641
`strong` element 1438
strongly typed language 196
Stroustrup, Bjarne 8, 439
`struct` keyword 661
structure 661
structured programming 2, 10, 11, 21, 85, 97, 140, 159, 171, 326
Structured Query Language (SQL) 896, 898, 905
structured systems analysis and design 11
style sheet 841, 1262, 1433
`sub` element 1387, 1446
sub-initializer list 266
subarray 261
subclass 125
`SubConcatination.cs` 648
`<subdialog>` tag 1242
submenu 522
`submit` attribute 1248
submit data to a server 1406
`submit` input 1406
subscript 238, 249, 266, 1387, 1446
subscription-based software 18
`Substring` method of class `String` 646
`SubString.cs` 646
Substrings generated from `String`s 646
subtraction assignment operator (`-=`) 121

Success property of **Match** 671
sum function 882
Sum.cs 147
summarizing responses to a survey 247
summary attribute 1234, 1459
summary tag 1338
Summation using **for** 147
Sun Microsystems, Inc. 1351
sup element 1446
superclass 125
supercomputer 3
superscript 1387, 1446
suspend a program 1313
Suspend method of class **Thread** 594
Suspended thread state 594
sweep 704
switch logic 156
switch selection structure 21, 98, 152, 155, 167, 170
SwitchTest.cs 152
Sybase, Inc. 897, 1351
symbol 1350
SyncBlock 599, 614
synchronization 599, 601, 607
synchronized 1150
Synchronized Multimedia Integration Language (SMIL) 1250
Synchronized.cs 607
synchronous error 440
syntax error 64, 66, 105
syntax error in HTML 1376
system caret 1263
SYSTEM flag 868
System namespace 62, 284, 633
system service 1109
System.Collections namespace 1160, 1185
System.Data namespace 26, 920
System.Data.OleDb namespace 920
System.Data.Sqlclient namespace 920
System.Drawing namespace 685, 686, 712
686
System.Drawing.Drawing2D namespace 685, 715
System.GC.WaitForPendingFinalizers method 317
System.IO namespace 760
System.Net namespace 1067
System.Runtime.Serialization.Formatters.Binary namespace 782
System.Text namespace 633
System.Text.RegularExpressions namespace 633, 669
System.Threading namespace 591
System.Web namespace 957
System.Web.Security namespace 1020
System.Web.UI namespace 957
System.Web.UI.WebControls namespace 957
System.Windows.Forms 69
System.Windows.Forms namespace 68, 69, 124, 147, 478
System.Xml namespace 847
System.Xml.Xsl namespace 882
SystemException class 448, 463

T

tab 68
tab character 62
tab order 1225
tab stop 68, 1225
Tabbed pages in Visual Studio .NET 560
tabbed window 37
TabControl class 560
TabControl properties and events 561
TabControl used to display various font settings 562
TabControl with **TabPage**s example 561
TabControl with **TabPages** example 561
TabControl, adding a **TabPage** 561
TabCount property of class **TabControl** 562
TabIndex property of class **Control** 485, 1230
table 897, 1231, 1232, 1375, 1398, 1431
table body 1459
table column 897
table data 1459
table element 265, 1399, 1459
table head element 1459
table in which record will be updated 918, 919
table of values 265
Table optimized for screen reading using attribute **headers** 1232
table row 1459
TableDisplay.cs 921
TabPage class 560
TabPage, add to **TabControl** 560
TabPage, using radio buttons 565
TabPages added to a **TabControl** 561
TabPages added to a **TabControl** 561
TabPages property of class **TabControl** 562
TabStop property 485
TabStop property of class **Control** 1230
tabular format 240
tag 950, 1338, 1375
tag name 1015
tag prefix 955
tail of a queue 1165
Tan method of class **Math** 182
tangent 182
target = "_blank" 1483
target = "_self" 1483
target = "_top" 1483
target="_blank" 1422
target="_parent" 1422
target="_self" 1422
target="_top" 1422
targetNamespace attribute 873
targetSessions attribute 1249
task 4
Task List window 1312
tbody (table body) element 1400, 1459
TCP (Transmission Control Protocol) 1108
TCP/IP (Transmission Control Protocol/Internet Protocol) 14
TcpClient class 1110
TcpListener class 1108, 1109
td element 1459
TEI (Text Encoding Initiative) 16

telephone system 1120
TemperatureServer Web service 1082
TemperatureServer.asmx.cs 1082
TempFileCollection class 852
temporary data storage 757
temporary value 114
termDigits attribute 1247, 1249
terminal 5
terminating right brace (**}**) of a block 213
termination 249
termination housekeeping 312
termination model of exception handling 442
termination phase 112
termination test 222
ternary operator (**?:**) 101
Test2 demonstrates polymorphism in Point-Circle-Cylinder hierarchy 400
Test3 uses interfaces to demonstrate polymorphism in Point-Circle-Cylinder hierarchy 424
Testing and Debugging Tip 12
Testing class **Cylinder** 370
text 736
text-based browser 1442
text box 1465
Text constant of enumeration **XmlNodeType** 849
text editor 64, 633, 1376, 1431
text element 1243, 1247
Text Encoding Initiative (TEI) 16
text field 69, 71
text file 847
Text property 48, 485, 493
Text property of class **LinkLabel** 531
Text property of class **MenuItem** 525
Text property of class **TreeNode** 549
TextAlign property 485
textarea 1406
textarea element 1408, 1465, 1466
text-based browser 1384, 1400
TextBox class 260, 475, 476, 488
TextBox properties and events 489
TextChanged event 489
TextReader class 760
text-to-speech (TTS) 1225, 1243, 1258
text-to-speech engine 736
TextToSpeech.cs 1226
TextureBrush class 690, 711, 715
TextWriter class 760
tfoot (table foot) element 1460
th (table header column) element 1232, 1459
The Diversity Directory 1297
The National Business and Disability Council (NBDC) 1297
thead (table head) tag 1400, 1459
this keyword 309, 313
this reference demonstration 311
this reference used implicitly and explicitly to enable an object to manipulate its own data and invoke its own method 310
Thompson, Ken 7
Thread class 592, 1118
Thread life cycle 593
thread life cycle 593
thread of execution 591
Examples
  Thread-priority scheduling 596
Thread-priority scheduling 596
thread-priority scheduling 596
thread scheduling 606
thread state 592
thread state *Dead* 593
thread state *Ready* 593
thread state *Runnable* 593
thread state *Running* 593, 596
thread state *Started* 593
thread state *Stopped* 593, 594, 627
thread state *Suspended* 594
thread state *Unstarted* 593
thread state *WaitSleepJoin* 594, 595, 600
thread synchronization 599
ThreadAbortException 593
ThreadPriority enumeration 594, 595
Threads sleeping and printing 596
threads sleeping and printing 596
ThreadStart delegate 593, 596, 599
ThreadTester.cs 596
three-dimensional application 717
Three-tier architecture 952
throughput 5
throw an exception 442, 446
throw point 442, 457
throw statement 451
Tick event of class **Timer** 580, 722
TicketReservation.aspx 1078
TicketReservation.aspx.cs 1079
Tic-Tac-Toe 1125
tightly packed binary tree 1177
TileHorizontal value in **LayoutMdi** enumeration 570
TileVertical value in **LayoutMdi** enumeration 570
time and date 580
Time class 22
Time1 abstract data type represents the time in 24-hour format 283
Time1.cs 283
timeout 1258
timeout attribute of **prompt** element 1241
Timeout property of **HttpSessionState** class 1003
timer 580
Timer class 720
timesharing 5, 11
timeslicing 594
title
  element 881
title bar 37, 46, 52, 1434
title bar string 147
title bar, MDI parent and child 567
title element 950, 1234, 1378, 1434
title of a document 1433
titles table of **books** database 898, 901
.tlb file 1368
ToggleKeys 1256
ToInt32 method of **Convert** 445, 446
ToLongTimeString method of structure **ToLongTimeString** 580

ToLower method of class **Char** 663
ToLower method of class **String** 648, 649
Tool tip demonstration 39
toolbar 38
toolbar icon 38
Toolbox 20, 963
Toolbox window 41
Tools menu 38
tooltip 39, 974
top 109, 116
top-down, stepwise refinement 3, 21, 109, 112, 116
top tier 953
ToString method of class **Decimal** 413
ToString method of class **String** 461, 650
ToString method of class **StringBuilder** 652, 654
ToString method of **Object** 349
ToUpper method of class **Char** 663
ToUpper method of class **String** 648, 649
Towers of Hanoi 234
tr (table row) element 1400, 1459
Trace class 1028
  Warn method 1028
  Write property 1028
trace element in a **Web.config** file 1028
Trace property 26
trace property of **Page** class 1028
trace.axd file 1028
TraceContext class 1028
tracing 1027
Tracing enabled on a page 1029
Tracing information for a project 1029
track 1250
tracking customers 987
trademark symbol 28
transaction-processing system 794 812, 816
transfer of control 96
Transform method of **XslTransform** 882
TransformTest.cs 882
TranslateTransform method of class **Graphics** 715
translation step 6
translator program 7
Transmission Control Protocol/ Internet Protocol (TCP/IP) 14
trapezoid 344
traverse a tree 1176
tree 26, 547, 1168
tree structure 841
Tree structure for **article.xml** 847
Tree structure for Fig. 18.1 847
Tree.cs 416
TreeNode class 549
TreeNode Editor 550
TreeNode Editor in VS .NET 547
TreeNode properties and methods 549
TreeTest.cs 1182
TreeTest.java 1174
TreeView 521
TreeView class 547
TreeView displaying a sample tree 548
TreeView properties and events 548
TreeView used to display directories 550
TreeViewDirectoryStructureTest.cs 550
trigger an event 476
trigonometric calculation 21
trigonometric cosine 182
trigonometric sine 182
trigonometric tangent 182
trillion-instruction-per-second computers 3
Trim method of class **String** 648
trim method of **string** 858
TrimToSize method of class **ArrayList** 1189, 1194
true 100
truncate 77, 114
truth 80
truth table 161
truth table for operator **!** (logical NOT) 163
truth table for the **&&** (logical AND) operator 161
try block 442, 446
try block expires 442
TTS (text-to-speech) engine 1243
Turtle Graphics 277
Tutor.cs 1098
two-dimensional data structure 1168
two-dimensional shape 685
two's complement 1284
twos position 1277
type = "hidden" 1405
type = "password" 1409
type = "radio" 1409
type = "reset" 1406
type = "submit" 1406
type = "text" 1406
type attribute 873, 1405, 1450, 1465
Type class 1230
type of a variable 75
typesetting system 633

U

U+*yyyy* (Unicode notational convention) 1352
UDDI (Universal Description, Discovery, and Integration) 1054
UDP 1108
UdpClient class 1120
uint primitive data type 196
uint promotions 194
ul element 1447
ulong primitive data type 196
ulong promotions 194
unambiguous (Unicode design basis) 1351
unary cast operator 114
unary operator 115, 163
UnauthorizedAccessException class 552
unbounded value 873
unchecked context 466
unchecked operator 466
Underline member of enumeration **FontStyle** 697
Underline property of class **Font** 696
underscore (_) 62
undo 38
uneditable text or icons 476
Unicode 73, 196, 197
Unicode character 758
Unicode character set 93, 634
Unicode Consortium 1351
Unicode Standard design basis 28, 1350, 1351
Uniform Resource Identifier (URI) 845

uninitialized local variable 109
unique class name 328
Univac 1108 14
universal (Unicode design principle) 1351
universal data access 17
Universal Description, Discovery, and Integration (UDDI) 1054
Universal Resource Locator (URL) 845
universal time format 284, 286, 288
UNIX operating system 5, 6, 7
Unload event 958
unmanaged resource 958
unnecessary parentheses 79
unordered list element (**ul**) 1388, 1389, 1447
Unstarted thread state 593
unstructured flowchart 170
Unsynchronized.cs 602
UPDATE 905, 918
UPDATE query 1075
UpdateCommand property of **OleDbAdapter** 927
822
updating a database 1405
upper-left corner of a GUI component 686
uppercase 62
uppercase letter 73
URI (Uniform Resource Identifier) 845
URL (Uniform Resource Locator) 950, 987
URL (Universal Resource Locator) 845
usemap attribute 1478
UseMnemonic property of class **LinkLabel** 531
user agent 1217, 1263
user control 1015
User Datagram Protocol (UDP) 1108
user-defined method 180
user-defined type 282
user interface 953
user interface event 219
UserControl class 579
user-defined exception class 462
userInput of class **_AgentEvents_CommandEvent** 750
ushort primitive data type 196
ushort promotions 194
Using a **PictureBox** to display images 507
Using an abstract data type 287
Using arrays to eliminate a **switch** structure 245
Using **CheckBox**es to change font styles 499
Using class **StackInheritance** 1162
Using default namespaces 846
using directive 62, 69
using elements of an array as counters 244
Using **For Each**/**Next with an array** 272
Using **GroupBox**es and **Panel**s to arrange **Button**s 496
Using inheritance to create a queue 1167
Using internal hyperlinks to make pages more navigable 1413, 1473
Using **meta** to provide keywords and a description 1478
using operator overloading 433
Using overloaded methods 223
using parentheses to force the order of evaluation 77
Using **RadioButton**s to set message-window options 502
Using **string** indexer, **Length** property and **CopyTo** method 636
using temperature and weather data 1087
using the **HugeInteger** Web service 1057
Using the mouse to draw on a form 510
Using the **Properties** window to set a property value 127
Using the **Stack** class 1195
UsingArray.cs 1186
UsingExceptions.cs 452
UsingFontMetrics.cs 699
UsingFonts.cs 697
UsingHugeIntegerService.cs 1057
UsingMDI.cs 571
UsingTabs.cs 562
UTF-8 1351
UTF-16 1351
UTF-32 1351
utility method 285
\u*yyyy* unicode format 1356

V
valid 865
valid identifier 62
Validate.cs 672
Validating user information using regular expressions 672
validating XML parser 865
validation service 1435
ValidationExpression property of class **RegularExpressionValidator** 979
ValidationTest.cs 874
validator 976
validator.w3.org 1376, 1435, 1451
validator.w3.org/file-upload.html 1435
Validators used in a Web Form that generates possible letter combinations from a phone number 977
validity 866
validity checking 298
valign = "middle" 1403
valign attribute (**th**) 1403, 1462
value attribute 1248, 1249, 1405, 1406, 1465
value of a variable 75
value of an attribute 1433
Value property of **HttpCookie** class 996, 997
Value property of **ServerValidateEventArgs** class 1020
value tag 1338
value types 196
VALUES 917
ValueType class 661
<var> tag (**<var>**…**</var>**) 1242
var attribute 1247, 1248
variable 72, 75, 282
variable name 75
variable reference 123
variable scope 144
variable size 75
variable type 75
variable value 75
variable, in previous statement 1317
variable, modify at run time 1317
Vault.com 1293
version 879

`version` attribute of `xml` declaration 1243
`version` parameter of `xml` tag 840
vertex 1478
vertical alignment formatting 1403
vertical coordinate 686
vi text editor 1376, 1431
ViaVoice 1217, 1235
video clip 591
video game 201
View menu in Internet Explorer 38, 69
`View` property of class `ListView` 554
Viewing the tracing information for a project 1029
`__VIEWSTATE hidden` input 986
`ViewStateEnabled` attribute 986
virtual directory 950
virtual key code 514
virtual memory operating system 11
`Visible` property 485
`VisitedLinkColor` property of class `LinkLabel` 531
Visual Basic .NET 19
Visual C++ .NET 19
Visual Inheritance through the Form Designer 577
visual programming 20, 478
Visual Studio .NET 20, 34
Visual Studio .NET Class View 333
Visual Studio .NET Debugger 27
Visual Studio .NET environment after a new project has been created 37
Visual Studio .NET-generated console application 65, 493
Visual Studio .NET menu bar 38
Visual Studio .NET **Menu Designer** 523
Visual Studio .NET menu summary 38
Visual Studio .NET Object Browser 333
Visual Studio accessibility guidelines 1218
Visual Studio creating and linking a virtual directory for the WebTime project folder 961
`VisualInheritance.cs` 575
`VisualInheritanceTest.cs` 577
vocabulary 16, 844
Voice Server SDK 1.0 1236
Voice settings window 1261
voice synthesis 1235
VoiceXML 25, 27, 1235, 1236, 1241, 1250, 1265
VoiceXML tags 1242
`void` return-value-type 186
volatile memory 4
Voxeo (`www.voxeo.com`) 1243, 1244
Voxeo Account Manager 1244
`.vsdisco` file extension 1057
`<vxml>` tag 1242

W

W3C (World Wide Web Consortium) 15, 27, 887, 1450
W3C host 15
W3C Recommendation 15, 872, 1431
W3C XML Schema 869
WAI (Web Accessiblity Initiative) 27, 1216
WAI Quick Tip 1216
`wait` element 600, 1249
`Wait` method of class `Monitor` 594, 607, 614, 615
`WaitForPendingFinalizers` method of class `GC` 317
waiting line 1146
waiting thread 615
WaitSleepJoin thread state 594, 595, 600
“walk” past end of an array 249
walk the list 1159
“warehouse” section of the computer 4
`Warn` method of `Trace` class 1028
WAV file format 733
Web 28
Web Accessibility Initiative (WAI) 27, 1264
Web-based application development 949
Web Content Accessibility Guidelines 1.0 1216, 1218, 1231, 1235
Web Content Accessibility Guidelines 2.0 (Working Draft) 1216
Web control 26, 949
Web controls commonly used in ASP.NET applications 966
Web controls demonstration 967
Web Form 26, 949, 988, 999, 1027
  `Properties` window 1028
Web Form page 949
Web Forms menu in the **Toolbox** 962
Web reference 1053
Web reference selection and description 1056
Web server 950, 1109, 1405, 1432, 1462
Web server/Client interaction. Step 1: The `GET` request, `GET /books/downloads.htm HTTP/1.0` 951
Web service 17, 26, 1041
Web Service Description Language (WSDL) 1042
Web-service method 1041
Web service that generates random equations 1095
Web Services 328
Web services located on `localhost` 1056
Web site 2
Web site using two frames: navigational and content 1420, 1480
Web user control 1015
`Web.config` namespace 1020, 1028
`WebClient` class 1085
`WebControl` class 957
`WebControls.aspx` 967
WebHire 1294
`WebMethod` attribute 1041, 1043, 1052
`WebService` attribute 1051
`WebService` class 1052
`WebTime.aspx` 953
`WebTime.aspx.cs` 955
`WebTime.html` 958
`Welcome.aspx` 1007
`Welcome.aspx.cs` 1009
`Welcome1.cs` 60
`Welcome2.cs` 67
`Welcome3.cs` 67
`Welcome4.cs` 68
well-formed document 865
`WHERE` 905, 906, 907, 908, 909, 911, 918, 919

`while` repetition structure 21, 98, 105, 109, 113, 114, 141, 143, 144, 167, 171
`WhileCounter.cs` 141
`White static` property of structure `Color` 689
whitespace character 62, 64, 650, 669
widening conversion 193
widget 475
`width` attribute 1383, 1441, 1442, 1459
width of text input 1406
width-to-height ratio 1442, 1383
Wiltamuth, Scott 9
Win32 API (Windows 32-bit Application Programming Interface) 8
window auto-hide 40
window gadget 475
window layout 38
window tab 37
Windows 2000 6, 27
Windows 32-bit Application Programming Interface (Win32 API) 8
Windows 95/98 60
Windows application 36, 60
Windows control library 581
Windows Control Library project 771
Windows Explorer 532
Windows Form 23, 476
Windows Form Designer generated code expanded 126
Windows Form Designer generated code reflecting new property values 128
Windows Forms proxy 1364
Windows Media Player 733
Windows Media Player demonstration 733
Windows menu 38
Windows NT/2000 60
Windows Registry 1363
Windows Wave file format (WAV) 733
Windows XP 6
WinForms 476
wire format 1043
wire protocol 1043
wireless application protocol (WAP) 1299
Wireless Markup Language (WML) 25
`WirelessResumes.com` 1299
Wirth, Nicklaus 10
WML (Wireless Markup Language) 25
word character 669
word processor 633, 643
Wordpad 1431
Working Draft 15
`WorkingSolo.com` 1300
workstation 6
World Wide Web (WWW) 3, 14, 15, 27, 475, 591
World Wide Web Consortium (W3C) 15, 27, 839, 887, 1264
`Write` method of class `BinaryWriter` 1119
`Write` method of class `Console` 66, 760
`Write` method of class `NetworkStream` 1110
`Write` property of `Trace` class 1028
`WriteByte` method of class `NetworkStream` 1110
`WriteLine` method 1313
`WriteLine` method of class `Console` 64, 760
`WriteRandomAccessFile.cs` 802
`WriteXml` method of `DataSet` 938
Writing records to random-access files 802
WSDL (Web Service Description Language) 1042
WWW (World Wide Web) 3, 14, 27
`www.advantagehiring.com` 1297
`www.advisorteam.net/AT/User/kcs.asp` 1297
`www.biztalk.com` 887
`www.careerpower.com` 1302
`www.chami.com/html-kit` 1376
`www.chiefmonster.com` 1301
`www.deitel.com` 2, 3, 29, 1376, 1439
`www.elsop.com/wrc/h_comput.htm` 29
`www.etest.net` 1297
`www.ework.com` 1300
`www.execunet.com` 1301
`www.jasc.com` 1441
`www.jobfind.com` 1295
`www.jobtrak.com` 1298
`www.microsoft.com` 29
`www.microsoft.com/net` 18
`www.mindexchange.com` 1297
`www.nationjob.com` 1301
`www.netvalley.com/intval.html` 29
`www.prenhall.com/deitel` 3, 29
`www.recruitsoft.com/process` 1297
`www.review.com` 1302
`www.sixfigurejobs.com` 1301
`www.unicode.org` 1353
`www.voxeo.com` (Voxeo) 1243, 1244
`www.w3.org` 15, 29
`www.w3.org/2000/10/XMLSchema` 873
`www.w3.org/History.html` 29
`www.w3.org/markup` 1431
`www.w3.org/TR/xhtml1` 1451
`www.w3.org/XML/Schema` 872
`www.w3.org/XML/Schema.html` 868
`www.w3schools.com/xhtml/default.asp` 1451
`www.webhire.com` 1294
`www.xhtml.org` 1451
`www.yahoo.com` 1381

X

x-axis 686
x-coordinate 686
Xalan XSLT processor 877
XBRL (Extensible Business Reporting Language) 25
Xerces parser 841
XHTML (Extensible HyperText Markup Language) 15, 25, 27, 1431
XHTML comment 1432
XHTML documentation of class `Circle` 1340
XHTML documentation of method `Area` method of class `Circle` 1341
XHTML form 1462

XHTML Recommendation 1264, 1451
XHTML table without accessibility modifications 1231
XLink (Extensible Linking Language) 16
XML (Extensible Markup Language) 15, 20, 25, 839, 1042
XML comment 1330
XML declaration 840
XML document containing book information 878
XML document referencing its associated DTD 867
XML document that conforms to a Microsoft Schema document 870
XML document that conforms to W3C XML Schema. 872
XML document that describes various sports 864
XML document that does not conform to XSD schema 876
XML documentation 1331
XML documentation comment 1330
XML documentation generated by Visual Studio .NET 1341
XML file containing **`AdRotator`** information 974
`.xml` file extension 841
XML file that does not conform to the Schema in Fig. 18.17 876
XML GL (XML Guidelines) 1235
XML Guidelines (XML GL) 1235
`xml` namespace 844
XML namespaces demonstration 844
XML node 841, 847
XML parser 841
XML processors 841
XML root 841
XML Schema 25, 844, 870, 872
XML serialization 1092
XML tag 840
XML to mark up a business letter 842
XML used to mark up an article 839
XML Validator 868
XML Validator displaying an error message 870
XML Validator used to validate an XML document 869
`XML.com` 888
`XML.org` 887
XML4J parser 841
XML-Data Reduced 870
`XmlDeclaration` constant of enumeration **`XmlNodeType`** 849
`XmlDom.cs` 851
`XmlNodeReader` class 848
`XmlNodeReader` used to iterate through an XML document 848
`XmlNodeType` enumeration 849
`xmlns` attribute 845, 847
`XmlReader` class 848
`XmlReaderTest.cs` 848
`XmlValidatingReader` class 873
`XMLWriter.cs` 939
`Xor` method of class **`BitArray`** 1496, 1508
XPath expression 857, 858, 890
XPath expressions and descriptions 865
`XPathNavigator` class used to navigate selected nodes 858
`XPathNodeIterator` Class 858
`.xsd` extension 872
XSD Schema document to which **`bookxsd.xml`** conforms. 872
`xsd:date` 873
`xsd:double` 873
`xsd:int` 873
`xsd:string` 873
`xsd:time` 873
`xsl`
  template 879
XSL (Extensible Stylesheet Language) 16, 844, 877
XSL document that transforms **`sorting.xml`** into XHTML 879
XSL specification 887
XSL style sheet applied to an XML document 882
XSL variable 882
XSLT (Extensible Stylesheet Language Transformation) 25
XSLT processor 877
`XsltArgumentList` class 882
`XslTransform` class 882
x-y coordinate 1418, 1478

Y

y-axis 686
y-coordinate 686
Yahoo! 1295
yellow 688
yellow arrow 1315
`Yellow static` property of structure **`Color`** 689
`YesNo` 148
`YesNoCancel` 148

Z

zero-based counting 143, 242
zeroth element 238